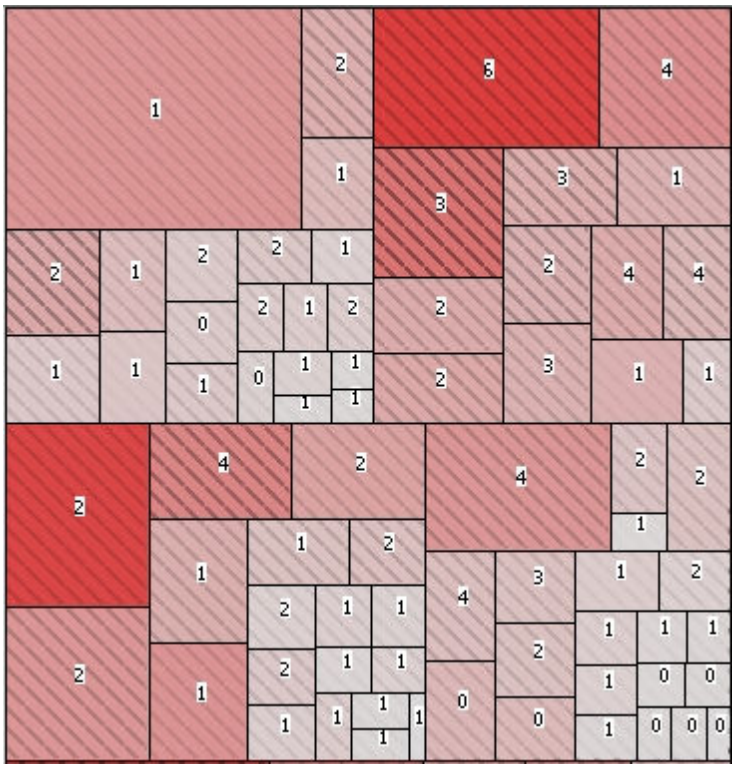
Static Program Visualization Within The ASF+SDF Meta Environment

Master Software Engineering

Qais Ali

Supervisor: Prof.dr. P. Klint



Centrum voor Wiskunde en Informatica

University of Amsterdam

Amsterdam, Holland

September, 2008

Contents

Preface

This thesis describes the graduation project which deduces a Master program Software Engineering at the University of Amsterdam. The project was to conclude new useful visualizations for the ASF+SDF Meta-Environment, which has been carried out at the Center for Mathematics and Computer Science (CWI) during the months of April - Augustus 2008.

The results of this research project would not have been possible without the help of many people. I would like to thank my supervisor Prof. dr. Paul Klint for his guidance and feedbacks. I am also grateful to all staffs of SEN department of CWI. I would like to thank P. Rekhtawan and A. Aldjaaf for reviewing my work, and my sweetheart Perzjin who motivated me. Finally, I would like to thank Hans Dekkers, Jan van Eijk, Hans van Vliet, Paul Klint, Tijs van Storm and Paul Griffioen for stimulating me during the master course.

Qais Ali
Den Haag

1 Introduction

Software comprehension is a crucial part of a number of software engineering tasks such as maintenance, reverse engineering and reengineering [1, 2]. In order to remove defects, enhance performance, add new features and leverage existing code, etc., programmers must first understand the existing code sufficiently to know what changes are needed, how to make them, and how to integrate code into new applications [3].

Software comprehension is a major factor in providing effective software maintenance and enabling successful evolution of software, since up to 50% of the total maintenance effort is spent understanding source code and other artifacts [4, 5]. It is even more interesting to note that the software maintenance process comprises more than 50% of development activities [6, 7] and 40-70% of the total cost of the project [8]. Furthermore, only well-understood software can evolve in a controlled manner [9]. Uncontrolled changes destabilize the project, leading to problems in keeping up with the schedule, or budget [10].

Software project time and budget are essential factors for project success. According to the Standish Group's Chaos report, only 16.2% of software development projects were completed on-time and on-budget and 52.7% of software projects will cost 198% of their original estimates. A considerable amount of this failure rate is due to incorrect or inconsistent software requirements. Another significant reason is the difficulty and duration of the maintenance process. As mentioned earlier, about 50% of software-understanding effort is spent in understanding processes. Hence, reducing program comprehension time will significantly reduce the development time and the project failure risk.

Software comprehension appears to be a complex problem-solving task [11] due to the large number of artifacts and their relationships within the software. Besides the intangibility feature, software is also unique in that it has both static and dynamic natures. In addition, understanding software requires the practitioner to draw on information scattered throughout the source code, which is a laborious task [12].

Regarding the importance of the time factor and the complexity of software comprehension there is a need to facilitate the program comprehension. Reducing this effort can be done by extracting the most significant information from the software system by a number of approaches such as abstraction, summarization, embellishment and transformation [12]. Obviously, most of the applications of these approaches such as syntax highlighting, metrics, refactoring and annotations belong to software visualization fields or can be supported by software visualization.

Software visualization, which is a type of information visualization, uses visual representations to make software more visible. It transforms information into visual forms that are easier to understand and remember and therefore can support a broad range of software engineering activities such as specification, architecture, detail design, construction, testing and maintenance.

Generally, software visualization visualizes three aspects of software: behavior, evolution and structure [13]. In this research, static program visualization is studied, by which the behavior and structure of software can be visualized. Visualizing structure will be the main area of the research while the focus will be only on the behavior aspects that can be visualized with static program visualization. Visualizing software evolution is outside the domain of this research since it is studied in a different research project at CWI by fellow student H. Baggelaar.

The goal of this research is **to investigate the existing visualization types, tools and techniques to find useful visualizations for the Meta-Environment** (the CWI's framework for language development, software maintenance and renovation). Meta-Environment is mostly used within two domains; design and implementation of Domain Specific Languages (DSLs) and software maintenance. To support understanding the process within these activities, Meta-Environment provides a number of visualizations such as a table-view. Studying results of analysis of large sized software in this table-view is annoying and time consuming, therefore, a better technique is required as an alternative to this table, which this research hopes to achieve. Software visualization is a young research area; its terminology is used inconsistently in many existing papers. To prevent the incorrect usage of this terminology, modern resources will be investigated to find the most accepted definitions. Furthermore, since a large number of software visualizations have been created in the last decade with different techniques and goals, selecting the right visualization for

a specific system without an overview of the current visualizations is difficult and does not guarantee the right choice. To avoid this, visualization techniques are classified, and then based on this classification, an overview of visualization is given, in which the techniques are evaluated according to specific criteria.
In addition to carrying out the main goal of this research, this document tries to answer the following questions:

- Why is program comprehension difficult?
- What is the visualization type?
- What is the visualization technique?
- How can software visualization fields be modeled?
- How can program visualization techniques be classified?

The next chapter is about background information, which provides an overview of program comprehension and visualization terms in addition to a new model for software visualization fields. The research contents is described in chapter 3. Chapter 4 is about visualization requirements and the limitations of the Meta-environment. An overview of 2-dimensional visualization techniques is given in chapter 5. Chapter 6 describes the two new visualizations, TreemapVis and TreeVis. These two new visualizations are experimented in chapter 7. Lastly, a conclusion is given in chapter 8.

2 Background and context

2.1 Program comprehension

Software comprehension is defined by Deimel & Naveda [14] as "the process of taking computer source code and understanding it". A better definition suggested by Russell Wood [12] defines software comprehension as "the process of gaining an understanding of how a software system functions". Program comprehension differs from software comprehension in that the latter comprises documentation beside the source code, while program comprehension refers only to the source code.

Most researchers agree with three models for describing comprehension strategies by the developers, namely top-down, bottom-up and the integrated model. In the bottom-up model, the comprehension task starts with reading the source code and repeatedly chunking the information until program understanding is reached. In the top-down strategy, hypotheses about the system functionalities will be formulated. Then these hypotheses will be validated experimentally. Repeating these two steps results in a hierarchy of hypotheses. This hierarchy can be proved by matching the hypotheses to the source code details. The top-down model is typically invoked during the comprehension process if the code or its language is familiar [3]. The integrated model is based on the combination of these two models. It has no concrete steps, because it depends on the developer's experience, program domain and the source code under investigation.

Program comprehension appears to be a difficult and time-consuming process, especially for large software systems. Hence, researchers and software engineers try to simplify it by using models, tools, and techniques that reduce the understanding time and effort. An example of these tools is a modern IDE such as Eclipse, which provides source code syntax highlighting, hypertext and program visualizations.

The significant comprehension effort is due to several factors. A number of these factors are listed below:

1. Software is intangible; the source code represents abstract objects having different characteristics with relations and interactions between them. In order to understand the software, an abstract impression of these concepts from the code has to be drawn. There is a need to know which objects are made from the source code and how they interact. This is quite hard, especially for concurrent software.
2. Software projects usually contain a large amount of code, especially large projects. Studying all these artifacts is an onerous process.
3. In the case of middle and large sized software, there is a large number of interconnected components. Studying these objects also requires a long time.
4. Software contains algorithms. Advanced and new algorithms are usually difficult to understand. Understanding algorithms takes a longer time compared to the normal code.
5. The human's memory is unable to save a large amount of software components and their interactions in a short time. Remembering these objects is essential for gaining insight into working software.
6. Software understanding involves addressing very specific issues such as searching for motivations behind method calls [15].
7. Software understanding involves reading code, asking questions, conjecture and search for the answer. Search in software comprehension is difficult and error-prone because people usually do not know where to look for information [16].
8. Understanding software takes more time if the engineer is unfamiliar with the software domain or the programming language.
9. Most of the software systems have few, or not up-to-date documentation. The analyst must search the source code for information, which takes a longer time.
10. Poor software quality makes software comprehension more difficult. For example, source codes that include long functions, bad variable names and code tangling are more difficult to study. Programs that have been developed without

coding convention involve several programming styles what makes understanding harder.

11. Programs that have been maintained over years are more difficult to understand due to the various programming styles, modified design and out-of-date documentation.
12. Programs that cannot be run due to bugs are hard to understand. The situation is more complex if it can be run with the bug, because it does not satisfy its specification.
13. During the comprehension process, the engineer may execute several comprehension tasks simultaneously such as reading, searching, thinking, translating, recall and mental modeling, which makes understanding harder due to focus problems [17].
14. Abstractions that are meaningful at the design level may become 'lost' at the code level [18]. Without abstraction, it is difficult to gain a quick overview of the software system.
15. Due to overusing abstractions, the code may become structured like a giant web [19]; the programmer may only be able to see a confusing network of interacting classes [20].

2.1.1 How program visualization facilitates software comprehension

Many papers claim that visualizations support cognition [21,22,23,24], the processes in human beings like perception, attention, learning, memory, thought, concept formation, reading and problem solving [22]. Visualization can support most of these fundamental processes. The book "Reading in information visualization: Using Vision to Think" [25] describes in detail how visualization can support the cognition process. This concept is generalized by M. Tory and T. Moller in a framework shown in Figure 2.1.

Method	*Description*
Increased Resources	
Parallel processing	Parallel processing by the visual system can increase the bandwidth of information extraction from the data.
Offload work to the perceptual system	With an appropriate visualization, some tasks can be done using simple perceptual operations.
External memory	Visualizations are external data representations that can reduce demands on human memory.
Increased storage and accessibility	Visualizations can store large amounts of information in an easily accessible form.
Reduced Search	
Grouping	Visualizations can group related information for easy search and access.
High data density	Visualizations can represent a large quantity of data in a small space.
Structure	Imposing structure on data and tasks can reduce task complexity.
Enhanced Recognition	
Recognition instead of recall	Recognizing information presented visually can be easier than recalling information.
Abstraction and aggregation	Selective omission and aggregation of data can allow higher level patterns to be recognized.
Perceptual Monitoring	Using pre-attentive visual characteristics allows monitoring of a large number of potential events.
Manipulable Medium	Visualizations can allow interactive exploration through manipulation of parameter values.
Organization	Manipulating the structural organization of data can allow different patterns to be recognized.

Figure 2.1: How visualization can support cognition [23, 26]

According to this framework, visualization affects the effort spent in search, recognition and inference processes. By grouping

associated information together in a compact display, large amounts of search can be avoided. Furthermore, applying structure to data increases accessibility and makes the hidden pattern obvious. In Figure 2.1 is the complete list of these factors. It is important to note that not all program visualizations are useful. Suitable visualization positively affects understanding while bad visualization can make it worse. Hence, it is very important to find the appropriate visualization.

2.2 Software and program visualization

Software visualization is a wide research area, involving a large number of visualization terms and techniques. In the past, a number of taxonomies have been proposed to categorize and classify software visualization terms, types and techniques. Despite these attempts, most of the papers and researchers disagree over the term definitions within software visualization. Nowadays, many visualization terms are used interchangeably; the term software visualization for example has many meanings depending on the author [27] and the definition of program visualization suggested by several papers are confusing with the definition of software visualization.

In this research, the two terms software visualization and program visualization will be distinguished, because software includes documentation besides the source code while the term program refers only to the source code. A definition of software visualization suggested by Diehl and another for program visualization defined by Price will be used.

Stephan Diehl defines software visualization as "the art and science of generating visual representations of various aspects of software and its development process". Software visualization can be seen as a specialized subset of information visualization, because software visualization is the process of creating graphical representations of abstract, generally non-numerical data, while information visualization refers to visualizing numerical as well as non-numerical data [17].

According to many papers, software visualization supports software engineers to cope with software complexity and reduces understanding time by supporting the cognition process during development and evolution of the software. However, in this research it is assumed that the software visualization must be suitable in order to be useful.

The other definition that will be used, depicts program visualization as "the visualization of the actual program code or data structures in either static or dynamic form" [28]. Considering these two definitions, it can be concluded that software visualization is a generic field that contains all visualization classes while program visualization is a specific part of this field.

Generally, program visualization focuses on improving program presentation and representation techniques. Program presentation refers to viewing the source code while program representation is the visualization of aspects of the code [24]. Source code highlighting and call graph are two typical examples of program presentation and representation. The ultimate aim of program visualization is to enhance the developer's understanding of various aspects of the source code under investigation. It can enable the software engineer to visually perceive program features that are hidden or difficult to obtain without the visualization.

2.2.1 A new model for software visualization fields

Program visualization will be the main field of this research; therefore, its position within software visualization needs to be investigated to find the usages and characteristics of its techniques. As mentioned earlier, there is no clear classification of software visualization fields. Therefore, based on a number of resources, especially Kang Zhang's papers, a new model for software visualization fields is introduced, as shown in Figure 2.2.

In this new model, a series of software engineering processes is shown. On the right side, a composition model of software visualization fields is depicted. The software engineering tasks are then mapped to the software visualization fields that can be used to support the task. For example, call-graphs that belong to the field static program visualization can be used in maintenance phase to support the software-understanding process.

In this model, software visualizations are classified into two fields: visual modeling and program visualization. The visual modeling field involves the visualizations that help software engineers in developing new software by reducing the time and complexity of engineering tasks. These visualizations are mostly used in the early phases of software development such as requirement

engineering or software architecture. The second field includes visualizations by which existing software systems can be analyzed. Hence, these visualizations are mostly used in the late phases of software development. Classifying software visualization into these two fields is attuned to the definitions of software visualization and program visualization suggested by Diehl and Price, in which program visualization can be seen as a kind of software visualization.

The first type of visualization represents mental work while most of the visualizations of the second type can be generated automatically from the source code. Each of these two fields consists of a number of subfields and each subfield belongs to one or more fields. In the next section, a brief description of these software visualization fields is given and the model is illustrated with a number of examples.

Concept visualization can be thought as the process of visualizing ideas, plans, analyses, processes, characters, and aspects of a system to model the system. Examples of this field are Gant charts, Petri nets, dataflow diagrams and state diagrams, which are mainly used in setting up the requirements and depicting the architectural concepts.

The visual modeling field contains a group of visualizations by which a developer can reduce the programming time. This group is mostly referred to as visual programming, which is the programming process through manipulating visual elements instead of written code. Nowadays, most of the IDE's include functionalities for visual programming, because the rapid development methods such as the Agile Development Method have adopted this technique. Typical examples here are Eclipse and Microsoft Visual Studio.

The visual programming group comprises two main types, visual query and visual GUI design. Visual query refers to constructing queries through visually related data entities and assigning query constraints such as in Visual Query Builder. Visual GUI design is the process of creating graphical user interfaces through visually collecting, adding and manipulating the interface objects such as buttons and text fields.

As mentioned, the program visualization is adopted to be the second main field of software visualization, which refers to graphical views or illustrations of the entities and characteristics of computer programs [24]. The program visualization field contains static and dynamic program visualization in addition to data visualization and program evolution visualization. It constitutes the main domain of this research, because the Meta-Environment is mostly used for software maintenance activities and these activities can only be supported by the program visualization field.

Data visualization is the graphical presentation of data, with the goal of supporting the viewer's understanding of the information contents. Data in this model refers to physical data kept in files or repositories. This data can contain information about the program such as an Rstore that contains software metrics. Visualizing data can also be used to support software project management tasks such as a traceability matrix.

Static program visualization is another type in the program visualization field, which refers to the process of visualizing aspects of a program without the need to run it. Examples of this field are call graph, class diagram, file explorers, and architecture diagrams. Visualizations of this type are used in several software engineering tasks. For example, use cases and sequence diagrams are widely used in discovering software requirements, class diagrams in designing software but also in maintenance activities. Hence, this visualization type falls into the program visualization field as well as the visual programming field. Furthermore, new tools exist nowadays by which the developer can visually construct the program such as tools for generating Java source code from a class diagram. Microsoft Robotic Studio and NetBeans Custom Code Generator are two examples in this field.

Static program visualization usually visualizes two aspects of software: structure and metrics. Software structure refers to the static parts and relations of the system, which includes the program code, data structures, relation graphs, and organization of the program into modules [13]. Software metric is an approach used for comprehension of software [18]. Based on this, it is possible to design a static visualization technique suitable for visualizing a combination of software structure and software metrics, which is a measure of some properties of a piece of a program.

Dynamic program visualization is the process of visualizing the dynamic aspects of a program with moveable elements, which represent events or communication between the system components such as in algorithm animation tools.

Dynamic program visualization is mainly used to support understanding of the behavior of the program, which refers to the execution of a program with real and abstract data. This execution can be seen as a sequence of program states, where each

program state contains both the code and the data of the program. Depending on the programming language, the execution can be viewed at a higher level of abstraction [13]. Dynamic program visualizations are mostly used as a support for understanding the execution of steps as well as the interactions between the program processes. They can also be used during the construction phases as debugging tools such as in Argus viewer.

The last field of program visualization is program evolution visualization, which is the process of visualizing changes in the program during the development time. Visualizing program evolution indicates changes, task distribution, software quality, correlation and maintenance difficulty [29]. Hence, it can be used in validation and maintenance tasks.

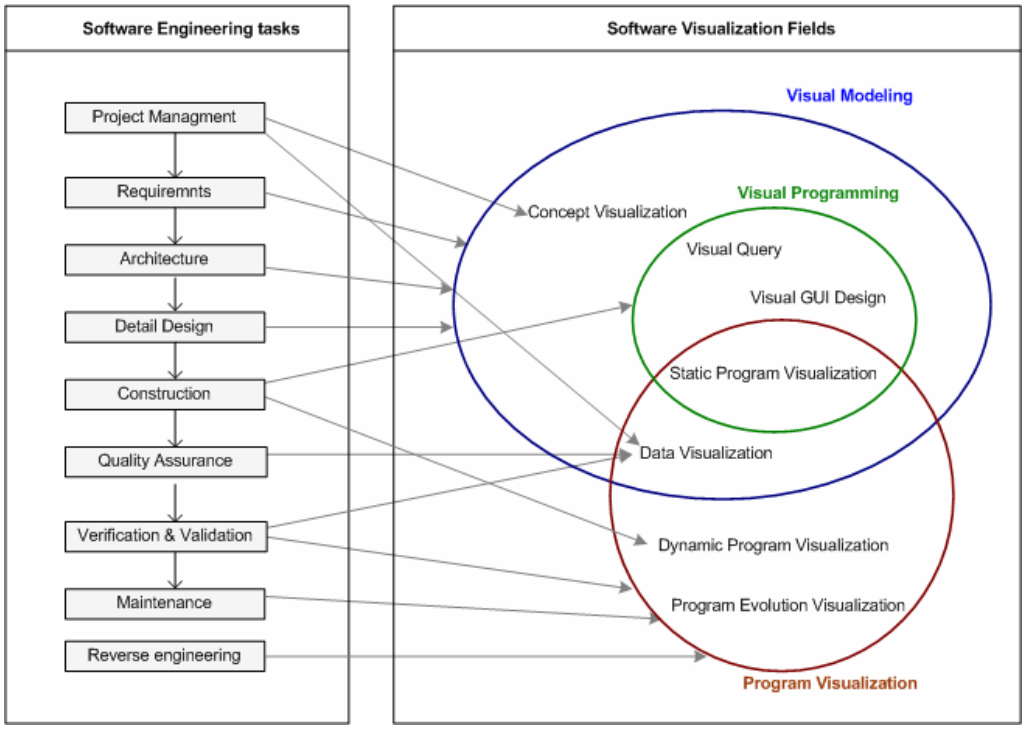


Figure 2.2: Software visualization fields used by Software Engineering tasks

2.2.2 Software visualization types

The visualization terms are used interchangeably in many resources, which leads to confusion in describing software visualization fields. Therefore, three definitions are specified in this research namely visualization types, visualization techniques, and visualization technique elements.

A set of visualization techniques that share fundamental display mode features can be organized under a specific group name. This group is called the visualization type. An example of a visualization type is 'static 2-dimensional nodes-and-links'. As the name says, it consists of one or more nodes linked with one or more links. This type includes a wide range of visualization techniques such as call graph, class diagram, and architecture diagram. Based on the definition of visualization type, a new classification of the existing visualization types has been introduced (see appendix B).

2.2.3 Software visualization techniques

Visualization techniques are forms of visualizations consisting of a collection of elements such as points, lines, shapes, texts, and textures. Each of these elements represents an entity or an attribute from the dataset that needs to be visualized. Each visualization technique belongs to a visualization type. For example, a class diagram can be classified under 'static 2-dimensional nodes-and-links', which consists of one or more nodes connected with one or more links.

In some cases, there are many visualization techniques to visualize a specific system. These visualization techniques can be of

different visualization types. A hierarchical system, for example, can be visualized using tree, treemap, and icicle plot (see Figure 6.1). The Navigation tree is of the type nodes-and-links while the rest are of the type space-filling according to our classification.

2.2.3.1 Fundamental elements of software visualization techniques

Visualization technique elements are visualizable parameters that affect viewer comprehension such as shape, size, color and texture. Any visualization technique is composed of a set of these basic elements that represent the software information. Selecting the appropriate set of these elements is essential for the success of the visualization because some of these elements are more effective in showing the differences between the visualization components, while others are more effective in displaying the properties of visualization components. An empirical study is performed by McGill to determine the accuracy level of visualization basic elements in quantitative tasks [30]. The model is expanded by Mackinlay with ordinal and nominal tasks, resulting in the model depicted in figure 2.2 [31].

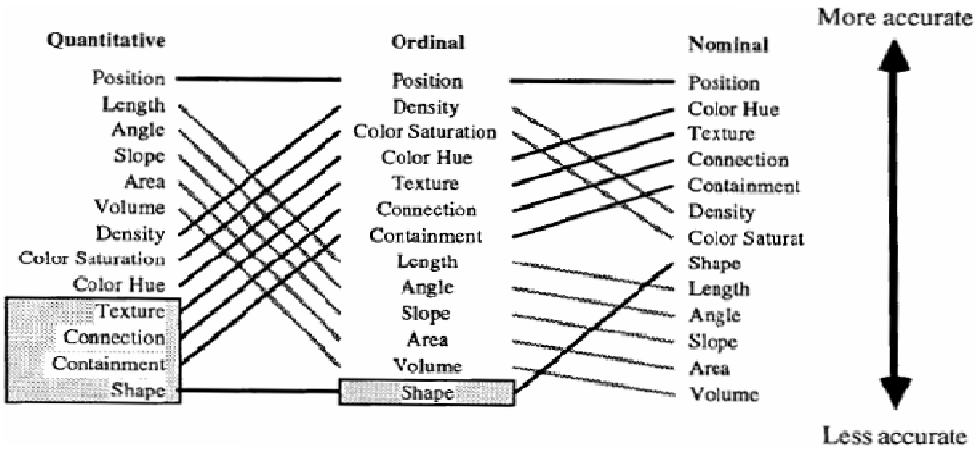


Figure 2.2: Ranking the perceptual tasks. The tasks shown in grey boxes are not relevant to their data types [31].

This model can be used for all visualization techniques including program visualization techniques. However, there are two other fundamental elements that are not taken into consideration in this model. These basic elements are orientation and text/value. Orientation can be used to show the difference between the elements or to improve the display and hence the observation. An example of orientation is top-down or bottom-up drawing of a tree. Information about software components can also be shown with texts in the visualization, which is widely used in software visualization.

2.3 Software visualization problems

Despite the benefits that many papers claim, software visualization is not yet a fully accepted part of the software engineering process in the software industry [76]. According to some studies, one of the main obstacles is the lack of integration of visualization into established tools, methodologies and processes for software development and maintenance. Another important problem of many existing software visualization methods and tools is their limited scalability with respect to the huge size of modern software systems and different datasets that need to be visualized [31, 32, 33]. These two issues must be taken into consideration during development of new visualization methods and tools.

2.4 ASF+SDF Meta-Environment

ASF+SDF Meta-Environment is a framework developed by CWI for language development, source code analysis and source code transformation. It consists of tools for syntax analysis, semantic analysis and source code transformation within an interactive development environment [34]. It has been successfully used in a wide variety of analysis, transformation and renovation projects. Furthermore, it is an open-source system that can be adjusted, tailored, modified, or extended.

2.5 Rscript, Rstore and basic relations

Rscript is a small scripting language based on the relational calculus. It is intended for analyzing and querying the source code of software systems: from finding un-initialized variables in a single program to formulating queries about the architecture of a complete software system [35]. The Rscript language has scalar types (boolean, integer, string, location) and composite types (set and relation). Expressions are constructed from comprehensions, function invocations and operators [36].

An Rstore (Relational store) is a language-independent exchange format, which can be used to exchange complex relational data between programs written in different languages [35]. It is a specific format for storing Rscript expressions and is used by the Meta-Environment for storing data of program analysis. More Information about Rscript and Rstore can be found in [35, 36].

2.6 Prefuse framework

Prefuse is an open source drawing framework written in Java, which can be extended with new drawing functionalities. The design of the Prefuse toolkit is based upon the *information visualization reference model*, a software architecture pattern that breaks up the visualization process into a series of discrete steps. This model is depicted in figure 2.3 where the source data is mapped into data tables that back the visualization. These backing data tables are then used to construct a visual abstraction of the data, modeling visual properties such as position, color, and geometry. The visual abstraction is then used to create interactive views of the data, with user interaction potentially affecting change at any level of the model [37]. In this way, it is easy to replace the graphical view with a new view, because the logic part is separated from the end graphical view. Using multiple views in the same application is possible too. In addition, Prefuse provides libraries for creating various data structures, supporting interactions, transformations and animations.

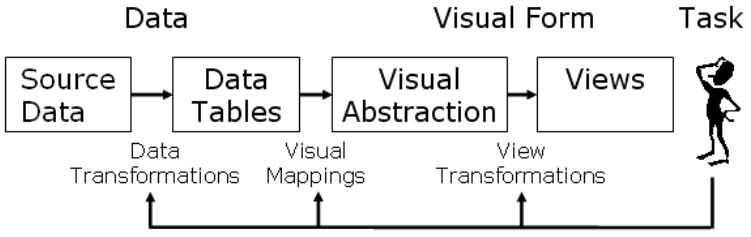


Figure 2.3: the Prefuse reference model

3 Research contents

3.1 Research motivation

Results of software analysis done in the Meta-Environment, are stored in Rstores. They contain relations build from fundamental Rscript types: integer, string, boolean and location. These Rstores can be complex and very large; depending on the size of the analyzed program, therefore understanding such Rstores is hard and time consuming.

As described earlier, the maintenance effort can be minimized by reducing the program comprehension time and difficulty. Reducing program comprehension effort is mostly done by using program visualizations. The Meta-Environment provides a number of visualization for some Rstore types. The type rel[str,int] for example can be visualized as bar-charts and pie-charts. Another visualization the Meta-Environment provides is the table-view, which is the only generic visualization that is able to visualize all kinds of Rstore types. Hence, the result of a software analysis may only be visualized using this table view currently.

Understanding Rstore-data displayed in a table-view is a long process especially for large Rstores. This process involves several activities such as reading, searching, reasoning, combining specific values, and constructing mental models. Performing these activities on the table-view takes long time. There is a need to find new visualizations that support understanding software analyzing results better than the table-view.

According to a number of studies, integration and scalability are two obstructions for acceptance of software visualization by the software engineering community. In the Meta-Environment case, integration and scalability of the visualization techniques are determined by the Rstore format and visualization GUI. Hence, there is a need to find a scalable Rstore format that can store generic analysis results. Finding such Rstore format requires studying Rscript and the Meta-Environment. Furthermore, this Rstore must be visualized in a generic way with a scalable GUI. Finding such a technique requires investigating the current available visualizations especially because standards for visualizations terms, methods and techniques are not yet established.

3.2 Research question

The purpose of this research is to investigate the existing visualization techniques and tools to find out useful visualization techniques for the Meta-Environment that can support understanding the results of program analysis stored in Rstores. The new visualization techniques have to function better than the existing table-view in its application domain. Based on this, the second step of this research is to validate the improvement using the new visualizations.

The new technique is not supposed to be a generic alternative to the existing table-view for each result of software analysis, because the table view is the most generic visualization technique while other software visualization techniques usually have smaller domain of application. However, it must be useful in the most use cases of the Meta-Environment.

3.3 Research plan

To find new advanced suitable visualizations for the Meta-Environment, the possibilities of using the elementary relation as a fundamental tool for visualizing abstract relationship between objects, needs to be studied first. The requirements, usages and limitations of the Meta-Environment have to be investigated too. On the other hand, the current existing visualizations types and tools have to be evaluated. Based on mapping between the results of these studies, a set of visualization techniques can be selected. Then, a couple of the selected techniques will be implemented and experimented to validate their benefits. These steps can be achieved follows:

1. Investigating the possibilities of using Rscript relations in modeling software visualizations. As mentioned earlier, the Rscript language consists of four scalar types related in two composite types. These composite types can represent the basic relations between the visualization elements. For example, a file or directory in software systems can be represented by their names, which can be implemented with a string. With a set of relations between these files,

dependencies between the files can be represented.

2. The possibilities and limitations of these relations have to be studied first. This can be carried out by creating a list of potential relations with their representations and techniques by which they can be visualized. Based on this list, decisions can be made whether there is a need to extend the Rscript language.
3. Investigating the existing software visualization techniques appropriate for understanding programs. This can be achieved by composing a list of the existing visualizations, in which a brief description, benefits and drawbacks of the technique have to be mentioned.
4. Studying the existing visualization tools, technologies and frameworks used in program comprehension field especially Prefuse, Eclipse visualization plugins, MOOSE and CodeSurfer. This step enriches the insight in visualization principles and extends the previous list. It also gives impression of various aspects and implementation issues of program visualizations.
5. A couple of visualization techniques will be selected from the list according to specific criteria. The detail criteria will be drawn up later according to usages, limitations and requirements of the Meta-Environment in addition to generic requirements for effective visualizations.
6. Implementing the selected visualizations within the Meta-Environment.
7. The impact of the implemented visualizations should be validated by a number of experiments.

3.4 Experimentation

Despite a large number of studies that has been performed on program visualization, little is known about the effects of such systems on learning [38]. Few researches can be found on the validation of program visualization. The possible reason is that many people think that all visualizations are beneficial.

To prove the advantages of the new visualization technique, there is a need to prove the ways visualizations support cognition. To do so, it is required to devise test questions according to the table 2.1. In such manner, the benefit of the new visualization can be guaranteed in addition to the estimation of improvement rate using the new visualization technique. The benefits of the new visualization will be proven by comparing the new visualization technique against the table-view. A table with search and sort functions such as Microsoft Excel or Open Office will be selected, because the Meta-Environment provides a functionality to export tables with software analysis result as CSV files. This CSV files can be viewed using these tools.

4 The solution visualization type

In this chapter, based on usages of the Meta-Environment and potential of Rstore, a type of visualization technique is selected. Then, based on resources a selection criteria of the visualization type is depicted.

4.1 The potential and limitation of Rstore

Results of software analysis within the Meta-Environment are saved in Rstores. These results may be the outputs of Rscript queries with different aims, resulting in different Rstore formats. Since the aim of this research is the visualization of these results, there is a need to investigate the possibilities for representing visualizations using Rstore relation types. For this reason, a list of potential Rscript relations has been created. For each of these relations, a set of representations that the relation can represent is given. Next, for each representation a set of visualization types or techniques is assigned by which the representation can be visualized. As an example, with rel[str,str] one can represent rel[file, file] by which software file dependency or file relations can be represented. The file dependency, for example, can be represented with various techniques such as directed graphs or undirected graphs. The complete list of these possibilities is shown in appendix A.

Considering the generality of Rstore and the fact that each program visualization technique consists of elements, element's properties and relations among these elements, it is expected that all static visualization techniques can be modeled using Rstore. However, since the Rscript language does not support the real type, modeling real values as properties of visualization elements is impossible. In some cases, this can be solved by normalizing the real values, but the problem remains in the case of an attribute set containing integers beside the real values. Nevertheless, from the constructed list in appendix A, it can be observed that almost all "2-dimensional static program visualizations" are modelable using an Rstore.

On the other hand, it can be observed that each program visualization technique has its specific domain. There is no program visualization as a generic alternative for tables because each result of software analysis can be visualized in table forms while no program visualization technique exists that can visualize all types of analysis results.

Within the Meta-Environment, there is no generic visualization technique, except the table-view, that can be applied for each Rstore, because the latter can have various formats. Hence, the most generic visualization technique must be selected to support the largest possible number of types of analysis results. Furthermore, the selected visualization has to be modeled in such a way that it can be applied for several visualizations.

4.2 The appropriate visualizations for the Meta-Environment

The Meta-Environment is mostly used in two domains, namely DSL (domain specific language) and software maintenance. The DSL domain involves development of languages for specific domain uses such as extending the COBOL language to develop special bank applications. The maintenance domain mainly involves the renovation of old and legacy systems such as renovation of software written in C.

The Rstores that are required to be visualized are the result of software analysis, which falls in the maintenance domain. Therefore, the focus will be on the visualizations that can be used in the maintenance process. The selected visualizations may also be suitable for language development, while it is not required.

Program visualization is the only field that supports the maintenance process (see Figure 2.1). Static program visualization, which is a kind of program visualization, can also be used in the DSL domain. Consequently, static program visualization techniques will be useful for the Meta-Environment. Incidentally, all the existing visualizations within the Meta-Environment belong to static program visualization as well.

The new visualizations must be programming language independent because the Meta-Environment is language independent. In addition, they must be suitable for visualizing each software system regardless of its programming language and style. Furthermore, the visualized systems can have different structures, components and sizes. Hence, the visualizations must be as

generic as possible and the Rstore format as open as possible. Furthermore, the systems that are being analyzed within the Meta-Environment can be very large, containing thousands of files. Therefore, the required visualizations must be suitable for visualizing large software systems.

The Meta-Environment currently provides a number of visualizations such as table and graph, which are implemented using the Perfuse framework, which is proven to be useful and reliable. To avoid the risk of new libraries and prevent implementation inconsistency, Prefuse will be used to implement the new visualizations. From this, the new visualization must also be implementable using this library. Since the Prefuse framework does not support drawing 3-dimensional visualizations, the new visualization must be 2-dimensional.

4.3 Software exploration tool as a solution

Based on the previous sections, the required visualization must visualize the common properties of the software systems written in various programming languages for different aims. The question hereby is 'which properties are common among all software systems?'

It is clear that each software system consists of a set of source-code files, organized in a specific file hierarchy. Each of these files has a specific set of properties, which may be represented as software metrics. Furthermore, almost all programming languages contain functions, which also have their own properties. Likewise, all software systems consist of relations between the files such as calls and dependencies.

From this, a suitable visualization will be one that links the visualization elements to the source code, in addition to graphically visualizing the common software features such as source-code file hierarchy, relation between the files and the properties of each file. This exploration tool, as it is called in some papers, meets the requirements for the needed visualization because:

- It can visualize all types of software systems, especially the old and legacy systems, since it is programming language independent.
- It can be modeled using the Rscript language.
- It can be generated automatically from the source code.
- It also helps the maintainer to form a mental model of a software system [39].

As mentioned earlier, openness and scalability are the two obstacles preventing industrialization of software visualizations. These two issues have to be taken into consideration in the course of designing a new visualization. For the exploration tool, the scalability problem can be partly solved by designing the tool in a scalable manner with respect to the types of features and relations that it can visualize. The exploration tool can also be an answer to the openness problem because it is applicable for all software systems.

Furthermore, the expected visualization tool is chosen to be static because firstly, most of the common features are static and can be represented in a static way such as file hierarchy and software metrics. Hence, the visualizations used in the maintenance phase are usually static. Second, dynamic visualizations can be considered as a support means for the second step in the analysis of large size software because it deals with visualizing the behaviors of the system and does not support the top-down comprehension approach. Many studies also claim that animations are no more effective than the equivalent static graphics due to the complexity or high speed of the animations [40, 41]. Furthermore, animations are very cognitively demanding [42] and "may draw learners' attention away from the more subtle but thematically more relevant features" [43]. Based on this we will only investigate the static visualizations in chapter 6.

A good framework for designing and evaluating exploration tools has been suggested by Storey & Margaret-Anne [39]. Though the framework was proposed to support a specific exploration tool called SHRIMP, it contains a good set of required properties for good visualizations. According to the author, the requirements mentioned in the framework were identified from a review of related literature as well as the author's observations from a series of user studies.

In this framework, reducing cognitive overhead is not assumed as a support for program comprehension, which is inconsistent

with the methods described in chapter 2. Nevertheless, the focus will be on the final requirements that the framework recommends. Since the framework is not generic enough, the visualization techniques do not need to satisfy all of these requirements. Therefore, the most relevant requirements are summarized as follows:

- In order to support bottom-up understanding, software visualization must provide immediate access to the atomic units in the program.
- Switching between different views results in a feeling of disorientation. To avoid this, multiple views of the program should be made available.
- The visualization should provide abstraction mechanisms to support the process of mentally building the hierarchical abstraction from the source code.
- It should provide searching capabilities to minimize the searching time.
- It should facilitate the integration of multiple information sources.
- It should indicate the user's current focus to minimize the risk of disorientation.
- It should have consistent views, sequences and terminology.

4.4 The selection criteria of the required exploration tool

From the previous sections, we can compose a list of requirements that the exploration tool must satisfy. This list shown below has been enriched with a set of desirable properties for good visualization suggested by Young and Munro [21].

1. The visualization must be suitable for visualizing the directory hierarchy of the software system.
2. The visualization technique should be able to visualize software metrics and the file source-code relations.
3. It is desirable to be able to link between the visualization and the original information, which represents the source code in this case [44].
4. It must be as generic as possible. The Meta-Environment is language independent, therefore the visualizations must also be language independent. It must be suitable for visualizing each software system despite its style and programming language.
5. It must be suitable for visualizing properties of large software systems. A software system may be very large containing thousands of files in a compact way. The new visualization must be able to visualize such software systems. It should effectively use the available display
6. It must be modelable using the Rscript language in a scalable manner.
7. It must be implementable using the Prefuse library. The current visualizations of the Meta-Environment use the Prefuse library.
8. It should present as much information as possible without overwhelming the user.
9. It should be well structured with low complexity. Well-structured information should result in easier navigation and low complexity trade offs with high information content.
10. It should provide different levels of detail. Granularity, abstraction, information content and type of information should vary to accommodate users' interests.
11. It should provide an approachable and flexible user interface. The visualization should include features to aid the user in navigating the visualization, for example using a search machine to reduce the user's chance of becoming lost. The user interface should be flexible and intuitive, and should avoid unnecessary overheads. Shneiderman (1996) suggests that: "a useful starting point for designing advanced graphical user interfaces is the Visual Information-Seeking Mantra: overview first, zoom and filter, then details on demand."
12. It should be suitable for automation- "A good level of automation is required in order to make the visualizations of any practical worth."

5 An overview of 2-dimensional static program visualization techniques

In order to select an appropriate technique for the required exploration tool, we need to study existing visualization techniques. In appendix B, a new classification of program visualization types has been introduced. Based on this classification, an overview of well-known program visualization techniques with a brief description is given in this chapter. Further, based on the criteria introduced in the previous chapter, it is decided whether the visualization technique can be used in the exploration tool. Based on this overview, a number of visualization techniques will be selected in the next chapter for further investigation.

This overview excludes the dynamic visualization technique due to a decision made in last chapter. It excludes the 3-dimensional techniques as well because the Meta-Environment currently does not support 3-dimensional drawing. Program evolution visualization is also excluded. Hence, this overview presents the state-of-the-art in the area of 2-dimentioal static program visualization techniques.

1. Nodes and links

1.1. Networks

1.1.1. Class blueprint

Class blueprint is a visualization of the inner structure of one or several classes. The main benefit of class blueprints is a considerable reduction of complexity when it comes to the understanding of classes. Class blueprints can quickly transmit the "taste" of a class to the viewer. Especially in the context of inheritance class, the visualization has proved to be useful, as the complexity the use of inheritance relationships can be visually perceived by the viewer [45].

Class blueprint is not appropriate as exploration tool within the Meta-Environment because it is only suitable for visualizing object-oriented software.

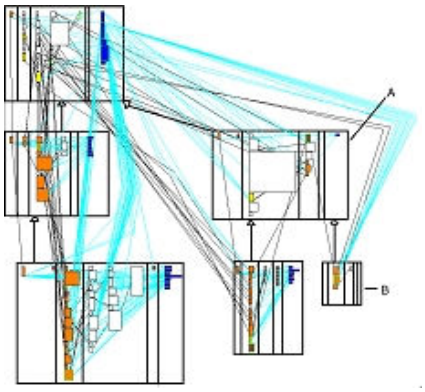


Figure 5.1: Class blueprint

1.1.2.SHriMP

SHriMP uses a nested graph view to present information that is hierarchically structured. It introduces the concept of nested interchangeable views to allow the user to explore multiple perspectives of information at different levels of abstraction.

It combines a hypertext following metaphor with animated panning and zooming motions over the nested graph to provide continuous orientation and contextual cues for the user [46]. SHriMp provides a good overview of the software, because it views the software aspects at several level of abstraction. Each level can be explored separately. However, it is not applicable for all software systems with different programming styles and languages; it is specialized in Java software systems.

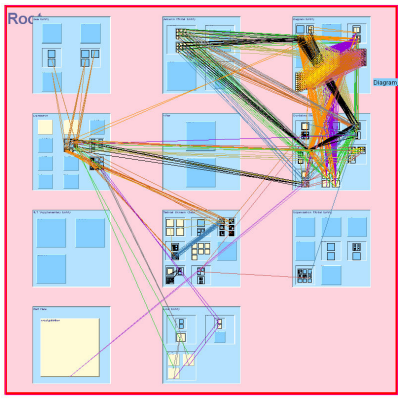


Figure 5.2: SHriMP

1.1.3. class diagram

Class diagram is a visualization technique for designing and analyzing object oriented systems. It shows the inheritance, polymorphism and containment among the software classes. A class diagram contains high amount of information on class abstraction level. The main drawback of class diagram is its specific domain and incompatibility in large software systems. The incompatibility problem can be solved using partial filtration of the system. Due to its specific domain, the class diagram is not applicable for our purpose.

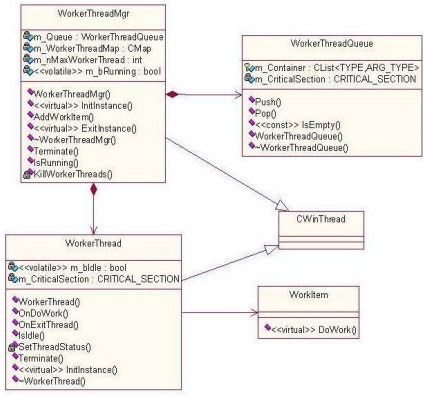


Figure 5.3: Class diagram

1.1.4. Beanome

Beanome is a visualization tool for aiding Component Based Engineering (CBE). The graph view displays components, their ports, and the connections among them. One important feature of Beanome is its capability to describe hierarchical definitions of components [47]. In other words, a component can be recursively defined as assemblies of other components. Like the previous visualization techniques, Beanome is unsuitable to use in generic exploration tools, because it is not suitable for different styles of software systems.

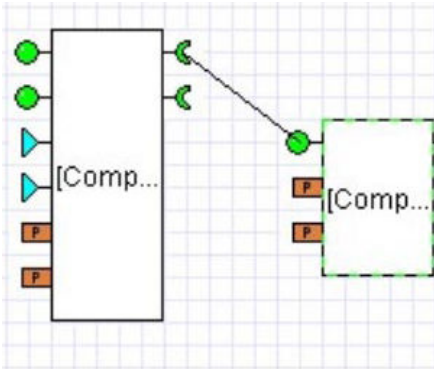


Figure 5.4: Beanome

1.2. Directed graph

1.2.1. Dependency graph / call graph

As its name says, a dependency graph shows the dependency among the software components. It is programming language independent, but it does not provide high amount of information because it represents only one aspect of the software. Dependency and call graph can easily be used as an additional layer on a 'nodes and links' visualization technique, because dependencies and calls can be between source code files. This additional layer will also indicate the coupling level.

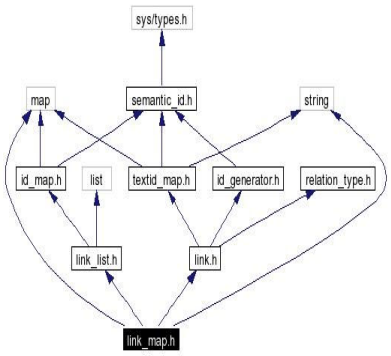


Figure 5.5: Call graph

1.2.2 Flow chart

Flow chart views the paths of a code fragment. The concept is generic because every programming language contains paths. In principle, flow charts are dynamic; they are mostly been generated without running the code. Flow chart representation lies in the code abstraction level and therefore not applicable for viewing large size software. It may be used in visualization techniques with different abstraction level from paths in the source code to the larger software components.

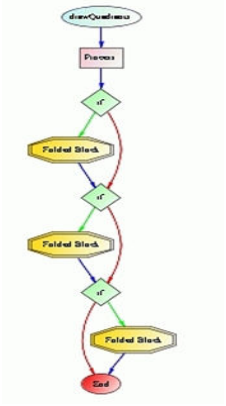


Figure 5.6: Flow chart

1.2.3. State diagram

State diagram is used to visualize process states and transitions among the states. State diagram may be suitable for visualizing the working of a part of the code. The main problem of state diagram is the impossibility to generate the diagram from the source code because it is difficult to model the abstract states from the source code.

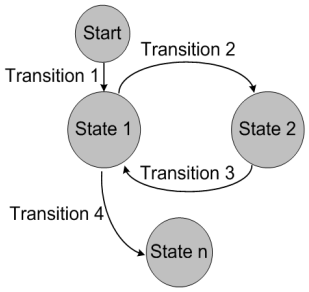


Figure 5.7: State diagram

1.2.4. Activity diagram

An activity diagram represents the business and operational step-by-step workflows of components in a system. It shows the overall flow of control. The main reason to use activity diagrams is to model the workflow behind the system being designed. Activity diagrams are also useful for analyzing use cases by describing which actions are needed to take place and when they should occur. It can also be used to understand the workflow of an existing software system but it has the same problem as state diagram.

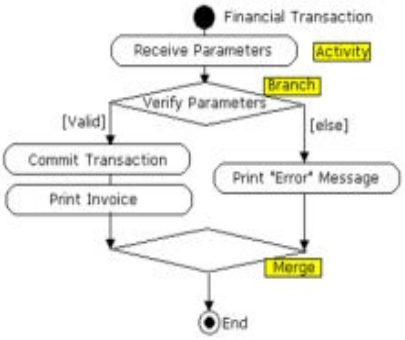


Figure 5.8: Activity diagram

1.3 hierarchy

1.3.1. Navigation tree

Navigation tree employs an expandable tree structure to represent the software file hierarchy, which uses folder icons to represent directories, and different icons to represent files [48]. Navigation tree can be used for all types of software systems, because every software system consists of files organized in file hierarchy. It can also be applied for visualizing class-methods-types or similar systems. Further, it provides filtrations support by hiding the hierarchy sub trees. Its drawback is that it consumes a relatively large display especially in case of large size software.

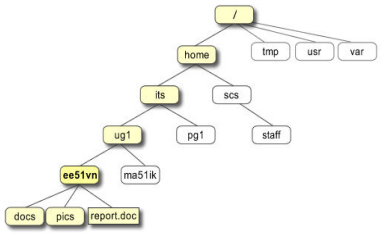


Figure 5.9: Navigation tree

1.3.2. Hyperbolic tree

Hyperbolic tree is a specific form of tree in which focus and interaction principles used to minimize the space needed to display the tree nodes. The scale advantage is obtained by dynamic distortion of the tree display. However, the amount of text that the hyperbolic browser displays is limited. In a 600x 600 pixel window, a standard 2-d hierarchy browser can typically display 100 nodes (w/ 3 character text strings) [49]. Hyperbolic tree solves the large display problem of navigation tree. However, it is not convenient for the user to visualize software that containing thousands of files, because a large number of nodes will be hidden. The second disadvantage is the impossibility to display all the nodes with their different features due to unsymmetrical size of the nodes. Hence, it is less suitable for viewing an overview of the software system.

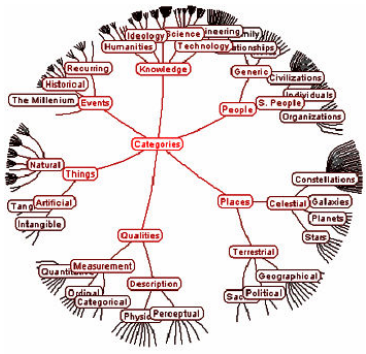


Figure 5.10: Hyperbolic tree

1.3.3. Circular tree

This type of hierarchical layout is comparable with the hyperbolic tree. It uses the available space more effectively than the radial tree layout while all the nodes of the tree are visible. However, it has the same problems as the hyperbolic tree for the large software systems. In addition, it does not support the focus and interaction as the hyperbolic tree. A circular tree with asymmetric node sizes requires more space than the normal one shown in Figure 5.11.

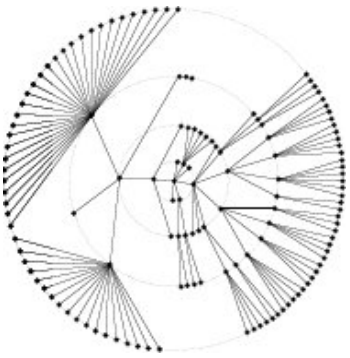


Figure 5.11: Circular tree

1.3.4. Balloon tree

Balloon tree is another tree layout in which the children are drawn in a circle centered at their parents and every sub-tree is repeatedly drawn as a new ring. The drawn algorithm ensures that two adjacent sub-tree sectors do not overlap. The 3-dimensional version of this layout is called Cone tree and therefore it can be obtained by projecting tree cones [50]. An advantage of Balloon tree is the visibility of all nodes. Nevertheless, Balloon trees that visualize large software systems without filtration and focus may lead to disorientation due to the large number of visible nodes. In addition, the deeper hierarchy level results in shorter links between a node and its parent. Consequently, it will be difficult to view a large file system having deep hierarchy level.

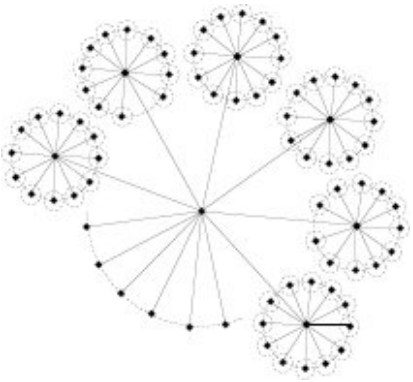


Figure 5.12: Balloon tree

1.3.5. Cheops

The Cheops technique is based on a compressed visualization of a hierarchical data set. The visual component for this method is a triangle [51]. In figure 5.13, a simple binary hierarchy is presented in its logical form above and its Cheops version under. Cheops is an effective way to show a large number of hierarchical nodes. Since the nodes overlap each other, it is hard to gain an overview of the system because gaining information about the nodes requires continuous clicking on the nodes.

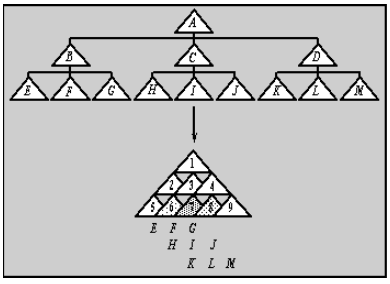


Figure 5.13: Cheops

1.3.6.. RINGS

Ringed Interactive-Navigation Graph System is a technique for visualizing large hierarchical data. It is based on a new ringed circular tree layout. Besides visualizing the topology of graphs, RINGS can also be used to visualize other information embedded in the graph such as file sizes. It can also locate specific files and types of files in a directory and reveal other aspects of the file directory structure [52]. One of RINGS's disadvantages is the small size of the leaf nodes which might be handled by zooming in the nodes. Another problem is the high complexity of the display, which can be handled by focus and interaction.

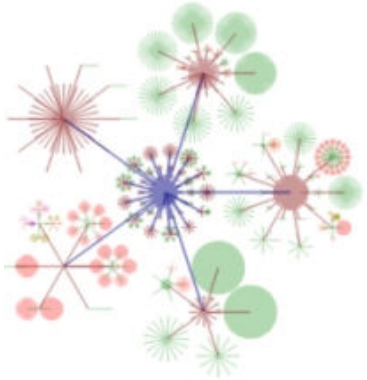


Figure 5.14: RINGS

2. Space filling

2.1. Treemaps

Treemaps subdivide the space into rectangles, which represent the nodes of the tree. Treemaps are efficient in using the display space; they can view up to thousands of nodes in a small display [53]. Further, it is programming language independent and it can be used for constructing software exploration tool. In the next chapter a number of treemap layouts will be handled, which can be constructed using different algorithms.

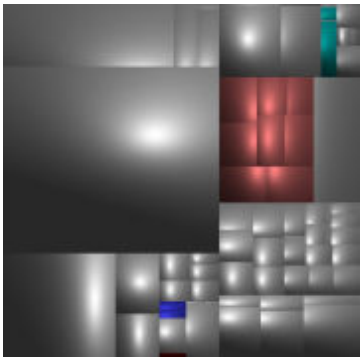


Figure 5.15: Treemap

2.2. Cropcircles

CropCircles is a technique to enhance user's ability to view the class structure at a glance. It is a hierarchy visualizer, and it uses containment to represent the parent-child relationship. According to the paper, CropCircle technique sacrifices the space-filling for better visual clarity, enhancing understanding of the topology [54]. The available space on a specific level is not used efficiently, which makes the node area on the next level much smaller. This makes perception more difficult than in a normal treemap. Due to tiny or hidden nodes, it is difficult to gain an overall overview of the system.

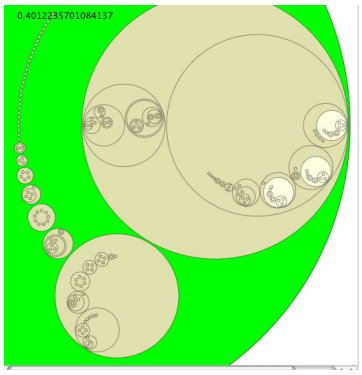


Figure 5.16: Cropcircles

2.3. Circular treemap

Using Cropcircles to represent the file hierarchy has some shortcomings. They do not fill the available space efficiently. They fill the available space to a varying degree which in the case of nested tree structures leads to the problem that circles of the same size could represent files (or folders) of a vastly different size [55]. Circular treemap solves this problem by drawing the sibling nodes beside each other according to their sizes in order to fill the available space. The node size is dependent on the hierarchy depth level; the deeper the hierarchy, the smaller is the node sizes, which makes it hard to gain an overview of the system.

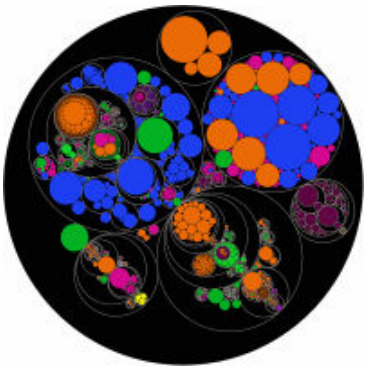


Figure 5.17: Circular treemap

2.4. Voronoi treemap

The approach of Voronoi treemaps eliminates the treemap's aspect ratio problems through enabling subdivisions of and in polygons. It allows creating treemap visualizations withi arbitrary shape areas, such as triangles and circles, thereby enabling a more flexible adaptation of treemaps for a wider range of applications [56]. However, viewing the system hierarchy is difficult with Voronoi treemap, because the sectors have different shapes and updating the dataset usually results in redrawing all of the nodes with new polygon forms, which lead to disorientation.

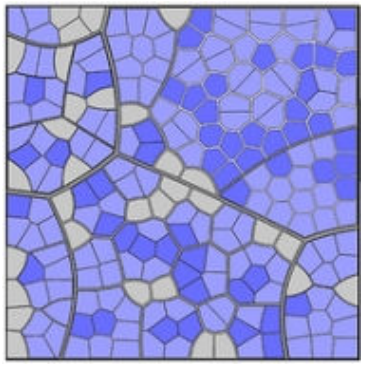


Figure 5.18: Voronoi treemap

2.5. Icicle plot

Icicle plot is a hierarchy layout in which the nodes are drawn as rectangles and the edges are implied by adjacency and spatial relationship. Like the tree ring, the Icicle plot displays the node size, which is proportional to the width of the node [57]. Hence, large hierarchy systems need horizontal and vertical scrolling. Further, the system hierarchy is better shown than the normal treemap, however, it dose not consume the available space efficiently because it concentrates only on one dimension of the available screen.

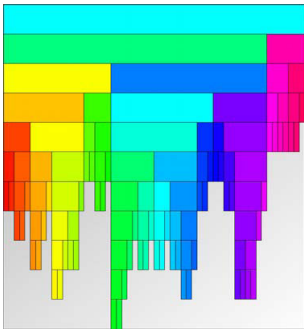


Figure 5.19: Icicle plot

2.6. Treering or Interring

Treering is also a space-filling visualization method. It is a radial version of Icicle trees described above. Node size is proportional to the angle swept by a node. Sunburst and Interring are Treerings with extra focus and context in which the details of expanded node are shown outside or inside the ring [58]. The effectiveness of the Treering is comparable with that of the Icicle plot. In addition, Sunburst and Interring have a better support for focus and interaction than Icicle plot.

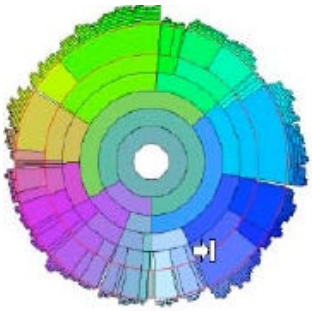


Figure 5.20: Interring

2.7. Seesoft

SeeSoft represents each line of source code as a single colored line on the screen. Lines are grouped in rectangles to represent each file. Thus, longer rectangles represent larger files. Line length indicates the length of the source code line and its color can be used to represent a variety of aspects, including the date of origination, date of change, id of the person who changed the line, which lines are bug fixes, nesting complexity, and the number of times the line was executed during testing [59]. This technique can be seen as an abstraction of the source code. It is useful to visualize specific features and metrics of pieces of the code. However, it needs a relatively large screen to visualize the complete code.

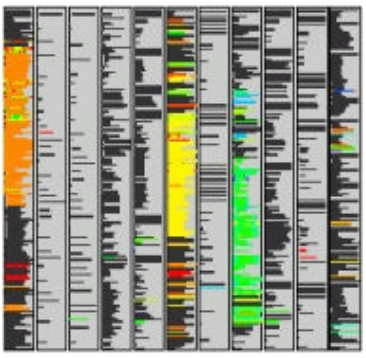


Figure 5.21: Seesoft

2.8. *2d Beam tree*

Beam tree is a relatively new visualization technique that is loosely based on treemaps. The root of the tree is made up of a long, horizontal rectangle. The children of the root are also rectangles, and are stacked on top of the root vertically. This process is continued until the entire tree is displayed. Some of the nodes may be aggregated together to reduce clutter. According to [60], 2d beam tree shows the structure of the hierarchy much better than treemaps. This is not correct for the large hierarchy systems because the nodes on the top of the hierarchy will be completely covered by the leaf nodes.

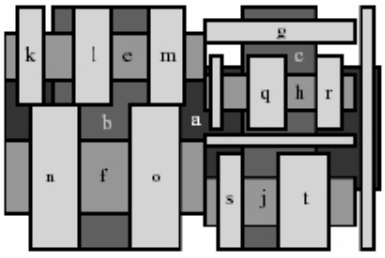


Figure 5.22: Beam tree

3 Matrix

3.1. Dependency structure matrix

DSM 'Dependency structure matrix' is a matrix where the rows and the columns are representing the software modules in two different dimensions and creating a cell. Each cell shows an intersection of two modules and a count of the dependencies between them. In this way, one can quickly recognize which other tasks are reliant upon information outputs generated by each activity. It can be generated from the source code such as in DSM plug-of IntelliJ IDEA.

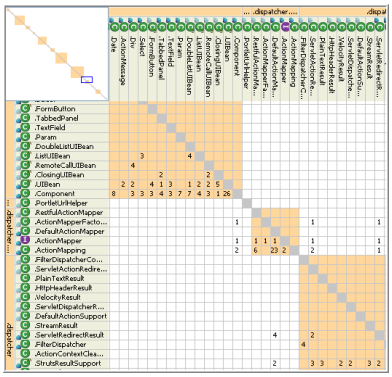


Figure 5.23: DSM

3.2. Inter-class call matrix

Inter-class call matrix provides an overview of the communications between couples of software classes by visualizing the calls between them. Classes are arranged along both axes in the order they are instantiated [61]. The links are shown with colored small rectangles, the color of the dots indicate the number of calls between the classes. Inter-class call matrix has zooming capability that allows the user navigate more detailed information on demand.

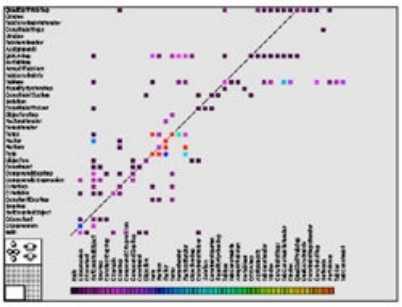


Figure 5.24: inter-class call matrix

4 Chart

4.1. Bar, Line and Pie charts

Bar and Pie charts are mostly used for visualizing distributions, ratio and the percentage of result values such as viewing the percentage of testing cover or software metrics. Line chart is usually used for viewing software development process such as visualizing developers contribution or increasing software LOC.

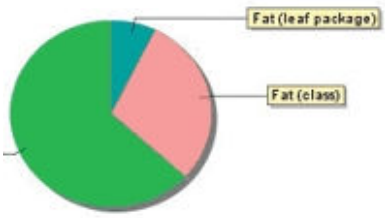


Figure 5.25: Pie chart

4.2. Kiviat

A Kiviat diagram is composed of axes extending from a central point. Each axis represents a data category such as software metrics and is scaled according to its data category. By using Kiviats, multiple items can be compared on the same diagram [78]. The advantage of the Kiviat diagram over the other charting / graphing techniques is that it can show more than two variables on one diagram .

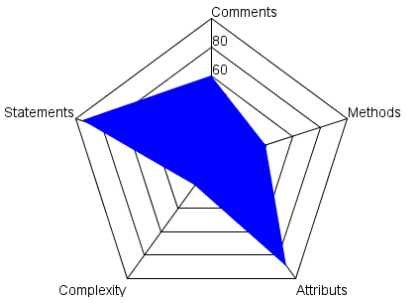


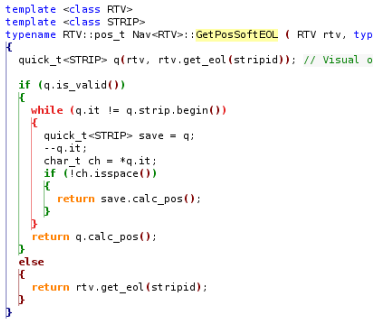
Figure 5.26: Kiviat

5 Textbased

5.1. Pretty Printing and hypertext

Prettyprinting is displaying the source code in different colors and fonts according to the category of terms. This feature eases writing in a structured language such as a programming language or a markup language as both structures and syntax errors are visually distinct.

In addition to prettyprinting, cross reference and tooltips are also used to support understanding the existing software codes.



```
template <class RTV>
template <class STRIP>
typename RTV::pos_t Nav<RTV>::GetPosSoftEOL ( RTV rtv, typ
{
  quick_t<STRIP> q(rtv, rtv.get_eol(stripid)); // Visual o

  if (q.is_valid())
  {
    while (q.it != q.strip.begin())
    {
      quick_t<STRIP> save = q;
      --q.it;
      char_t ch = *q.it;
      if (!ch.isspace())
      {
        return save.calc_pos();
      }
    }
    return q.calc_pos();
  }
  else
  {
    return rtv.get_eol(stripid);
  }
}
```

Figure 5.27: Pretty printing

6 Shapebased

6.1. Structure101

Structure101 generates the architecture of the software system from the source code with a certain abstraction level, which can be adapted on demand. It supports understanding existing software systems and it helps software development team to evolve the code in line with the intended architecture [62]. Structure101 is available as a plug-in for Eclipse and IntelliJ IDEA.

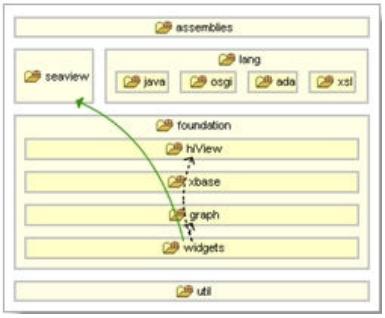


Figure 5.28: Structure 101

7 Combinations

Not all of the combination types of visualization techniques do exist as tools that can be used in the software comprehension process. Hence, the rest of this list includes only the existing visualization techniques.

7.1. Nodes and links - Spacefilling

7.1.1. Elastic hierarchy

Elastic hierarchy attempts to solve the tree and treemap problems by combining the space efficiency of treemaps with the structural clarity of node-link diagrams [63]. Hence, it can be an effective solution for huge software systems.

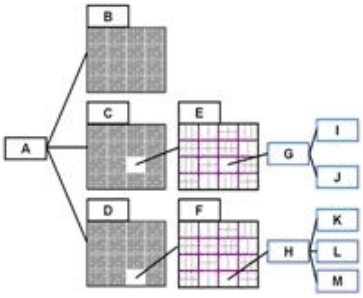


Figure 5.29: Elastic hierarchy

7.1.2. Calls inside treemap

In this technique, the hierarchical structure is displayed as a treemap and the adjacency edges as curved links. The links are depicted as quadratic bezier curves that show direction using curvature without requiring an explicit arrow. Some graphs that can be meaningfully visualized as an underlying tree structure with overlaid links [67]. However, figure 5.30 shows that this technique suffers from visual clutter as well when many links are visualized [64].

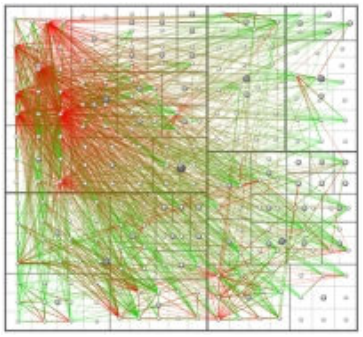


Figure 5.30: Calls inside treemap

7.2. Nodes and links – Matrix

7.2.1. Matrix browser

Within the Matrix browser, different hierarchy results of the information extraction are displayed as the two axis of an adjacency matrix. In the center of the matrix the selected type of association between leaves of the hierarchy trees are shown as interactive symbols, which allow the user not only to navigate the trees, but also the associations [65]. The user can increase or reduce the displayed amount of information by expanding and collapsing the tree nodes. The selection nodes on the axes can also be replaced by filtering the information space using different kinds of queries.

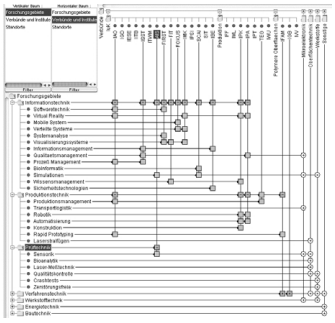


Figure 5.31: Matrix browser

7.3. Nodes and links – Charts

7.3.1. Metricsview

MetricView is essentially an UML visual tool that adds highly customizable metric visualizations to well-known diagrams. Any element can have any number of metrics, which can freely be chosen for any elements. Metric data and UML diagrams are provided as a separate input files to the MetricView. MetricView can visualize class, sequence, state, use case, and collaboration UML diagrams and supports boolean and numeric metrics [66]. It can be observed that MetricView can only be applied to object-oriented software.

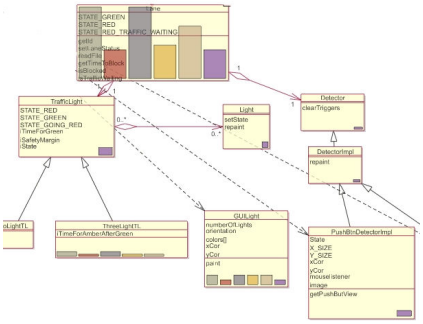


Figure 5.32: Metricsview

7.3.2. Kiviat graph

Kiviat graph includes Kiviat diagrams as nodes that represent software components such as modules or classes. These nodes can visualize multiple metric values of n development releases. The graph also shows coupling dependencies among the modules, which also indicate the coupling strength. Kiviat graph is mostly used for visualizing the evolution of the system [32, 78]. It highlights the good and bad times of an entity and facilitates the identification of the entities and relationships with critical trends. It also represents potential refactoring candidates that should be addressed first before further evolving the system.

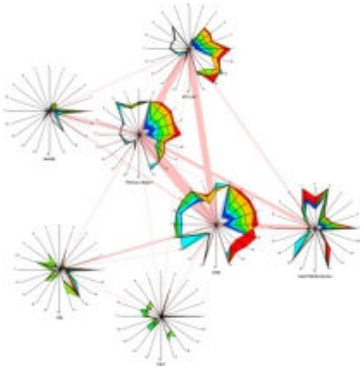


Figure 5.33: Kiviat graph

7.3.3. Hierarchical edge bundles

This approach uses the path along the hierarchy between two nodes having an adjacency relation as the control polygon of a spline curve; it manages edge congestion by interactively curving edges away from the point of focus, which opens up sufficient space to disambiguate the relationships. Thus, it visualizes the software file hierarchy and the call graph between the hierarchy nodes that represent source code files [68]. Hierarchical edge bundle is useful for gaining an overview of the system.

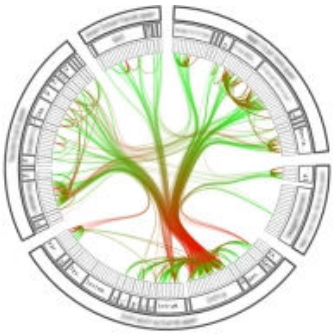


Figure 5.34: Hierarchical edge bundles

7.4. Nodes and links – Text based

7.4.1. RELO

RELO is en Eclipse plug-in for exploring source code inside the software class diagram. It focuses on abstraction and interaction to visualize java code. It helps developers explore and understand large codebases [69]. It also allows developers to zoom in to view details and edit the code using embedded text editors. The diagrams represent only a small manageable part of the code and do not include irrelevant details, thus allowing a developer to focus on important code relationships.

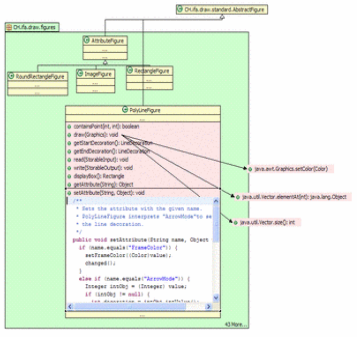


Figure 5.35: RELO

7.5. Space filling – Chart

7.5.1. pixel matrix

A pixel-matrix is a small rectangle, which has one or more pixels. The color of these pixel-matrixes varies based on a measured value, which represents one data item. It is used for visually analyzing multi-attribute data by encapsulating colored pixel-matrixes into charts, tables, and time series for visualizing data patterns, correlations, exceptions, and evolution in long-running time series data [70]. This approach differs from the existing techniques by encapsulating regular graphics with color-encoded data values. Therefore, users can analyze data and detect root-causes without clicking through many charts and listings.

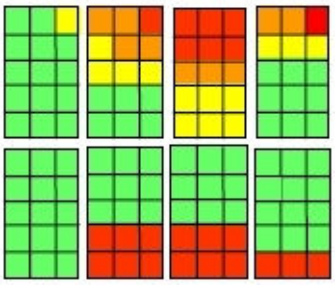


Figure 5.36: Pixel matrix

7.6. Matrix – Chart

7.6.1. ZAME

Zoomable Adjacency Matrix Explorer (ZAME) is a visualization tool for exploring networks at a scale of millions of nodes and tens of millions of edges. ZAME presents an adjacency matrix graph representation aggregated at multiple scales. It allows analysts to explore a graph at many levels, zooming and panning with interactive performance from summary to the most detailed views [71].

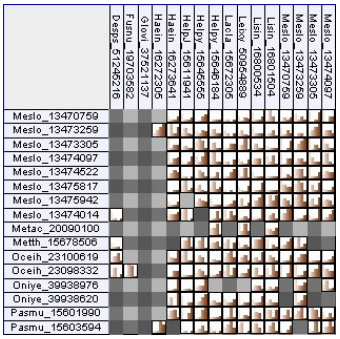


Figure 5.37: ZAME

6 The new visualization techniques

In chapter 4, an exploration tool as a type of required visualization technique has been suggested. For this exploration tool a visualization technique is required, by which a software file hierarchy can be visualized. In this chapter, the required visualization technique is selected. Using this technique the exploration tool is designed and implemented.

6.1 Selecting the appropriate technique for displaying file hierarchy

Based on the overview given in chapter 5, there are a number of visualization techniques that can be used for visualizing hierarchical file systems with the required exploration tool. The suitable techniques are shown in Table 6.2 in which they are evaluated according to the criteria depicted in 4.4.

In Table 6.2, it can be observed that all suitable visualization techniques are generic since they are programming language independent. Furthermore, they are suitable for automation, since they can represent software file systems that can be drawn automatically. Likewise, they are all integratable with software source code since the nodes can represent the software files containing the source code, which can be visualized beside the software file hierarchy. Furthermore, they are modelable in a scalable manner using Rstore, since the same Rstore can be used as a dataset for all of them in which variable names and values can be left for the end user to define.

Concerning the ability to visualize file system hierarchies, the navigation tree scores the best (++), because it provides the best hierarchical view compared to other techniques, while the hierarchy is not clear (-) in Cheops and Cropcircles. The Hyperbolic tree is the least effective (--) in viewing file properties and relations among the nodes, since leaf nodes are hidden and viewing additional relations makes the visualization complex in the case of large software. In addition, it is impossible to gain an overview of the node properties due to the size difference of the nodes. Regarding the capability to visualize large software systems in a compact manner, the normal and Voronoi treemaps are the best applicable visualization techniques, since they use all of the display pixels.

Treemap is a very compact visualization technique- up to 3000 nodes can be displayed easily in an image with a dimension not greater than 640 x 480 pixels [72]. In the Treemap, the file system hierarchy cannot be clearly displayed, especially if the directories are hidden. By decreasing the space of the leaves and using that space for the directories, viewing the hierarchy can be improved. Furthermore, treemap has a relatively low complexity since the leaves are not overlapping each other, as in the RINGS. The unfavorable drawback of treemap is the unsuitability to merge it with other visualization components, such as placing a link layer on the top of Treemap to view the relations between the leaf nodes (see figure 5.30).

The Voronoi treemap has the same capacity as the normal Treemap due to its use of all of the display pixels. Its advantage over the normal treemap is the ability to be drawn on a non-rectangular display area. Using the Voronoi treemap is beyond consideration since the Meta-Environment provides a rectangular space for the visualizations and the system hierarchy display is not clear due to unsymmetrical node sectors. Likewise, the hierarchy system represented with Cropcircles is not clear. The nodes in the deeper levels are small or cannot be shown, which leads to drawing problems since Prefuse draws the visualization objects per pixel and therefore zooming out does not help. Furthermore, the size of the nodes is reserved for showing the hierarchy; thus, it cannot be used to show node properties. For these reasons, Cropcircles is simply not suitable for our purpose.

The Circular treemap can be thought of as an improvement of Cropcircles in which the available space is better utilized. The node sizes can be used to represent the node properties in addition to color, texture and text. Still, the available space in a normal treemap is more efficiently used while the nodes in this model are better recognizable. Since we strive to visualize large systems, the normal treemap is preferred over the circular treemap.

The greatest disadvantage of the Icicle tree is the inefficient use of the available display since it grows in one direction, which requires scrolling the display to view all the nodes in the case of large software systems. Furthermore, representing the node properties is less appropriate here compared to the Navigation tree, due to the unsymmetrical size of the nodes. In contrast, the

Navigation tree has a good, well-known hierarchical view, which supports abstraction and filtration. Its disadvantage lies in its incompact display, where not all of the files can be shown in the same display. Another shortcoming is the awkwardness of using the area of non-leaf nodes to represent the property of the node.

Displaying the leaf node properties can be achieved by showing the difference in node area size, color and texture. Displaying these attributes is well applicable on the treemap and navigation trees while it is less appropriate with the tree ring and Icicle trees because the leaf nodes are visually separated from each other in the treemap and the navigation tree in contrast to the rest.

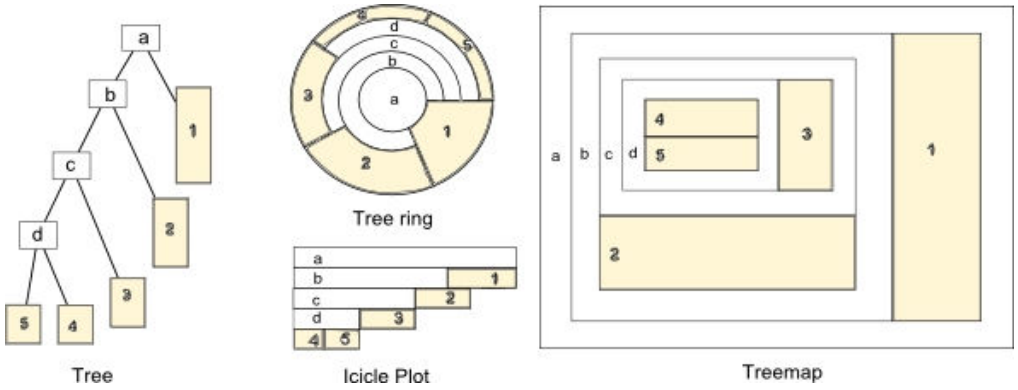


Figure 6.1 A number of ways to visualize a hierarchical data structure. The leaf nodes drawn in light yellow nodes represent the files in the hierarchy while the white blocks represent the directories.

Elastic hierarchy combines the tree's clarity of the system hierarchy with the compactness of treemaps. It would be practical if the only directories that do not contain other directories be drawn as a treemap, otherwise the directory view will be inconsistent and confusing. As with the navigation tree, an elastic hierarchy supports filtration of parts of the system, however, it fails to use all the available drawing space such as in the normal treemap.

The Balloon tree is a tree-like visualization that can view directories besides the leaf nodes in rounded forms, which makes it more compact than the normal tree. The problem with the balloon tree is in the case where the nodes have a large number of children, such that the ring becomes large and the leaf node so small that it will be difficult to gain a clear overview of the system. For the same reason, it will be difficult to display the different properties for these leaf nodes simultaneously. Another disadvantage of this technique is the inability to hide the directories, which is also the case with the Icicle plot.

RINGS is another compact visualization that can view the directories beside the leaf nodes. It is an improvement to the balloon tree where the sibling sub-trees are drawn in an overlapping manner, in which interaction is used to focus on a specific ring. This makes RINGS more compact than the balloon tree and the navigation tree; however, it makes the display complex. The nodes having a large number of siblings are difficult to distinguish from the others (see Figure 5.14). Another obstacle to using RINGS is the complexity to implement with Prefuse. Prefuse renders the visualization using basic elements such as lines and nodes. In this case, focusing on a group of nodes requires additional effort.

Tree in circular level and Balloon tree have the same problems as the hyperbolic tree. For the large software systems, the display becomes large and the nodes so small that it will be difficult to obtain an overview of the software. The first provides a clear hierarchy level, by which a number of node attributes can be used, such as color, texture and text, but the node size has a limited use.

Cheops is an extreme compact visualization wherein the nodes are represented with triangles, by which groups of interior nodes are represented. Because of the overlapping nodes, it is hard to gain an impression of the system. Since most of the nodes are hidden, the user needs to inspect the nodes to gain an overview. Hence, it is not a suitable technique for visualizing a software system. It is efficient for the case in which compactness is the only important point.

The Hyperbolic tree also has a compact visualization. Its characteristics are mostly the same as a tree in circular levels, while it

better supports interaction and filtration. The focused node within a hyperbolic tree has the greatest size relative to the other nodes. The size of the nodes decreases in the direction from the focused node to the leaves. In the optimal case where a leaf node has been focused, the majority of the leaf nodes obtain the smallest size, which is not preferable.

The Tree ring and interring are circular translations of the Icicle tree wherein the leaves are within the circle. The Tree ring is not the optimal way for visualizing a file system, since the file will be represented with the cells on the outside of the ring with different sizes and heights, which are difficult to compare with each other in terms of size. Furthermore, hiding the directories represented with the interior nodes does not help in clarification of the files since resizing the leaf nodes leads to damage of the hierarchical information. As a result, the space-use is not efficient and the capacity of viewing leaves is lower than treemap.

A 2-d beamtree is a derivation of the treemap, in which the nodes are rectangles drawn on top of each other according to their hierarchy. Children of each node are drawn as rectangles on top of their own parent horizontally or vertically in opposite direction to the parent. The 2-d beamtree is unsuitable to use in the exploration tool, because the hierarchy is not clear for large software, and the leaf nodes have a low aspect ratio, which is inappropriate for displaying leaf properties. In addition, it leads to confusion in the size of the leaves.

A Hierarchical edge bundle is an interesting visualization, which combines the hierarchical file system with the links among the files. It contains a higher degree of information compared to other techniques discussed in this section, which makes it suitable for gaining a quick global overview of the system. Again, viewing thousands of files in a clear manner requires larger space compared to treemap, since the interior space is reserved for displaying the links among the files. In addition, it may be difficult to draw it using Prefuse, because the nodes have arc shapes.

The aspects discussed in this section are summarized in the next table:

	Ability to view hierarchy systems	Ability to view metrics and relations	High information content	Low visualization complexity	Varying levels of detail	Approachable user interface	Compact visualization of large software	Implementable using Prefuse	Modelable with Rscript in scalable manner	Integration with source code	Generic programming styles	Suitability for automation
Treemap	+	+	+/-	+	+/-	+	++	+	+	+	+	+
Voronoi treemap	+	+	+/-	+	+/-	+	++	+	+	+	+	+
Crop circle	+/-	-	+/-	+/-	+/-	+/-	+	+	+	+	+	+
Circular treemap	+	+	+/-	+/-	+/-	+/-	+	+	+	+	+	+
Icicle tree	+	+/-	+/-	+	-	-	+/-	+	+	+	+	+
Elastic hierarchy	+	+	+/-	+/-	+/-	+	++	+	+	+	+	+
Navigation Tree	++	++	+	+/-	+	+	-	+	+	+	+	+
Balloon tree	+	+/-	+	+/-	+	+	+	+	+	+	+	+
RINGS	+	+/-	+/-	-	+	+	+++	+/-	+	+	+	+
Cheops	+/-	+/-	+/-	-	-	-	++	+	+	+	+	+
Tree in circular level	+	+/-	-	+	-	+	+/-	+	+	+	+	+
Hyperbolic tree	+	--	+/-	+/-	+/-	+	+/-	+	+	+	+	+
Tree ring/ Sunburst	+	+/-	+/-	+/-	+/-	+	+/-	+	+	+	+	+
2d beam tree	+	+	+/-	+/-	-	+	-	+	+	+	+	+
Hierarchical edge bundles	+	+/-	++	+/-	+	+/-	+	+/-	+	+	+	+

Table 6.2: Evaluation of hierarchical visualization techniques according to criteria depicted in 4.4

From the previous paragraphs, it is clear that Treemap has the largest capacity for displaying a huge number of nodes in a fixed display size while the node properties can be better shown compared to many techniques discussed in this section. This is why Treemap has been chosen to be used in the preferred software exploration tool. The lack of hierarchical clarity in Treemap can be handled by using an interaction technique, by which the hierarchy can be shown or hidden on demand. The main disadvantage of Treemap is the awkwardness of viewing the links among the nodes, which makes the Treemap more complex. Since the aim is to display the maximum amount of nodes in a fixed display size, viewing the links among the nodes is considered as a secondary requirement. However, the links can be partly shown on demand by clicking on the node.

For evaluation purposes, it will be useful to select a second technique from the nodes-and-links category to help compare the benefits and drawbacks. The Navigation tree will be used for this purpose. Furthermore, as mentioned earlier, an important property of visualization is to associate the visualization objects with the source code, by which the exploration process will be on a multiple abstraction level. It is possible to view the source code in a separate window beside the main visualization display by which the files represented by the treemap can be associated with associated nodes and their source code.

In addition, highlighting can be used to display the links of a specific file represented by a Treemap node. In order to display the properties of the source code belonging to a specific file, an abstract overview of the code can be displayed such as in Seesoft (see Figure 5.21). Each line of the code is represented with a color line that indicates the property value of the associated part of the code. This overview will be used to provide a global overview of the code and can be seen as an intermediate view between the file node and its source code.

6.2 Design

In the previous section, Treemap and Navigation tree have been selected to be the main part of the required exploration tool. In this section, the appropriate Rstore format and treemap algorithm are selected and the exploration tool is designed.

6.2.1 The Rstore format

As mentioned earlier, customizability and scalability appear to be the major problems of software visualization tools. These two problems form a barrier to employing software visualizations in the business domain, which forces us to deal with them during the development of visualization techniques.

In order to create a generic visualization technique, the dataset behind the visualization technique must be made as customizable and scalable as possible. In our case, where Rstore represents the dataset, customizability can be achieved by decomposing the Rstore structure into two parts: required and optional. The required part should represent the fundamental component needed to build the visualization structure, which includes the system hierarchy and the nodes. The optional part of the dataset should represent node properties and their association, such as software metrics belonging to each file node in a file hierarchy system.

An excellent feature of Rstore is its scalability with respect to input size and type. Rstore can include three composition types: tuple, set and rel that enable scalability. For example, defining a data format as a tuple of <string, int> provides the possibility to fill in any software metric with its value. Building an Rstore set from these tuples as set[<str,int>], which is the same as rel[str,int], provides the possibility to fill in any set of metrics with their values.

Furthermore, Rscript provides a specific type loc that represents a specific location in a given file. This loc type as described in 4.4 can represent a complete file as well as a specific fragment such as functions. A software file system can be represented with a set of locations, because the location consists of the absolute path to all files in that system. In this way, the required part of the Rstore type definition and relations between files can be represented with a set of locations.

The set of software metrics belonging to each file can be represented with a set of relations between the name of the metric, its value and set of location that represent the functions in the file with their values. The relations between the files and the set of software metrics will be optional in the Rstore. Hence, a set of locations that represent the software file system is enough to build the basic structure and functionalities of the visualization. In addition, to be able to visualize the same software system with both

new visualizations, both visualizations should be represented with the same Rstore format.

The Rstore type definition will be: rel([loc,set(loc),rel([str,int,rel([loc,int])])])

The first location represents the given file in the system; set(loc) is optional, it represents the set of files which are linked with the first file. The last part rel([str,int,rel([loc,int])])]) is also optional; it represents the set of software metrics for the first file as described above. It is important to recall that a relation in Rscript is a set of tuples, so each relation word in this definition refers to a set of the given type.

The following are a number of example inputs of this Rstore type definition:

rel[file_paths]: can represent the software hierarchical system. The hierarchical information can be obtained from the paths

rel[class_paths]: the existing classes in a software file system. Files can include multiple classes.

rel[function_paths]: a set of the functions in a software file system.

rel[file_paths, set(file_Paths)]: can represent the software hierarchical system with the existing links between the files

rel[file_paths, rel([file_software_metric, metric_value])]: software hierarchical system with a set of software metrics and their values per file.

rel([file_path, set(linked_file_paths), rel([file_software_metric, metric_value, rel([function_paths, metric_value])])]): the same as above with file links and sets of software metrics with their value per file.

The last example is an optimal input for the new visualization tool; therefore, we will use it as a type definition for the test Rstore in this research.

6.2.2 The information flow

Rstore is used as an intermediate medium to store the result of software analysis. The content of these results may be different based on the aim of the analysis. To be able to visualize these different results with the new visualizations, the Rstore must satisfy the format described above. To do so, the maintainer must first decide which set of software metrics he needs. He must also decide whether the relation between the source code and the files is needed, then the data has to be extracted from the source code and stored in an Rstore that is used for creating the two visualizations.

In figure 6.3, two visualization operation arrows are shown which represent the needed input information. The constructed Rstore is required to form the basic structure and functionalities of the two visualizations while the source code is used for viewing the optional highlighted source code, which is linked to their source file representations.

Figure 6.3: information flow

6.2.3 The visualization model

The visualization model is based on the Prefuse reference model shown in figure 2.3. The extracted information from the source code is stored in the Rstore. This Rstore dataset is parsed to create a table object containing the same information. Each row represents a file or directory while the columns represent attributes and properties of the file. These attributes contain the file name, type, parent, children and a number of optional properties of the source code file such as metrics and dependencies.

The next step is to create tree structure data from this table that corresponds to the software hierarchical file system in which the nodes preserve their properties. The size attribute of the nodes is determined recursively from the bottom to the top of the tree, which is used for drawing the nodes. The tree data structure and the file properties are used to create the graphical views. Based on this tree data structure, the treemap and navigation tree will be created as end graphical views. Displaying the source code

overview beside the basic exploration technique is an optional function; therefore, the Rstore dataset is sufficient for visualizing the basic part of the software system.

The two graphical views are interactive, where each node is rendered separately. The user interaction can affect the last level of the visualization process by which the whole view or specific parts will be updated such as in the case of exchanging the default node attribute, which leads to repainting all the view elements.

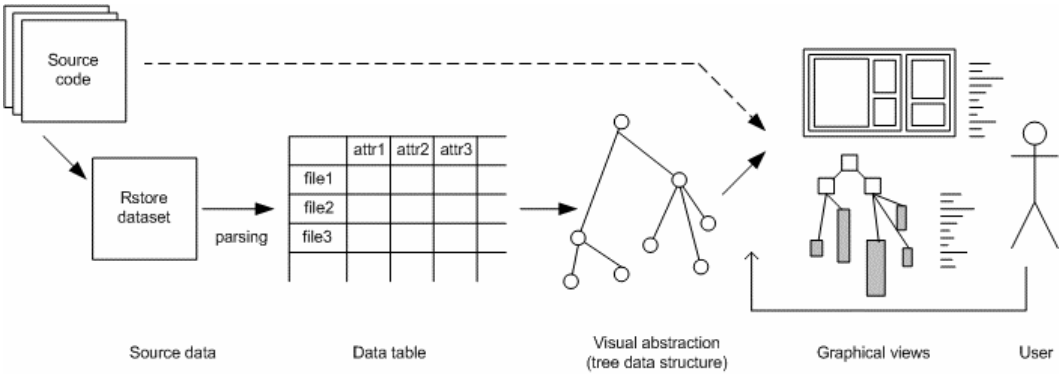


Figure 6.4: visualization model

6.2.4 The appropriate treemap algorithm

In the previous steps, Treemap has been selected to be a part of one of the visualization techniques. The next step is to select the study of the available algorithms by which a Treemap can be drawn. There are several algorithms for drawing a readable Treemap, resulting in several types of Treemap such as Slice-and-dice, Squarified and ordered Treemap.

The Slice-and-dice algorithm recursively partitions the planar display area along both dimensions. It uses parallel lines to divide the rectangle representing an item into smaller rectangles representing its children. These parallel lines are drawn vertically and horizontally based on the depth level of the hierarchy. According to [73] large hierarchies, greater than 1000 nodes, can be displayed without congestion using the slice-and-dice algorithm. However, this is only correct for lucky cases because the slice-and-dice algorithm often creates layouts that contain many rectangles with a high aspect ratio (height/width or width/height based on the initial rectangle). The worst case is a system with a large number of nodes in a single hierarchy level. Due to the large number of high aspect ratio rectangles, the slice-and-dice layout has a readability issue.

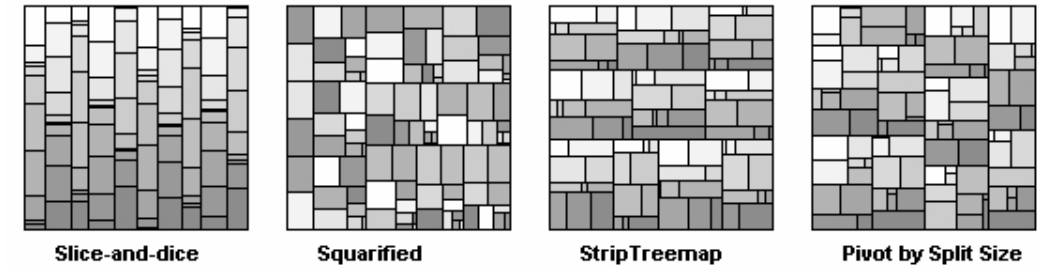


Figure 6.5: four different layouts of treemap (assembled from [74])

A Treemap squarification algorithm has been introduced by Eindhoven University. It subdivides the Treemap into rectangular-areas, such that the resulting rectangles will have the lowest aspect ratio. “These rectangles use space more efficiently, are easier to point at in interactive environments, and are easier to estimate with respect to size” [35]. A squarified treemap solves the slice-and-dice problem because a large number of nodes in a specific hierarchy level does not affect the Treemap’s readability.

According to the paper [74], squarified Treemaps suffer from some drawbacks. The first drawback is that changes in the data set can cause dramatic discontinuous changes in the layouts. The second issue is that a squarified Treemap does not provide support

to recognize ordering patterns, which many data sets contain. This is why two new algorithms have been introduced, resulting in two new Treemaps: Strip and Pivoted Treemaps. The Strip algorithm is similar to the squarified Treemap algorithm; the difference is that the strip algorithm strives to arrange the rectangles according to other criteria besides achieving a low aspect ratio for each node. The Pivoted Treemap algorithm is a new algorithm type that aims to improve the stability problem of strip Treemaps.
According to a study that has been performed to compare the readability and stability of this type of Treemap layouts [75], ordered Treemaps are more readable and more stable than squarified Treemaps. Nevertheless, the squarified Treemap is more applicable for the case of this research for two reasons. First, the Rstore dataset of the visualization does not change for software during the visualization phase. Hence, the layout does not change due to changes in the dataset. Second, changing the selected software metric results in a dramatic change in the layout in the case of ordered Treemaps. Each selected software metric has its rectangle value distribution and ordered algorithms will draw the Treemap according to this distribution. Thus, switching between the software metrics results in changing the node positions, making the search time much longer. Hence, the squarified Treemap layout was selected to be combined with a text-based display that displays the source code.
The Squarified treemap is based on a specific subdivision algorithm, by which the nodes of the Treemap are optimally squarified. An overview of the algorithm is shown in Figure 6.6, in which the squarification mechanism is depicted for seven sibling nodes. Initially, the nodes must be organized in an ascending list based on the normalized weight used to determine the size of the node. The nodes are consecutively added to the initial rectangle. For each node, its ratio aspect is examined vertically and horizontally. Based on this, the row or column containing the examined node is rearranged.

Figure 6.6: squarified treemap algorithm (from [53])

Rearranging the rows and columns is determined by the function *worst(),* which constitutes the crucial part of the function *squarify()* . The function *worst()* gives the highest aspect ratio of a list of rectangles, given the length of the side along which they are to be laid out. It determines whether an appended node fits in the current column/row or has to be replaced to the next row/column.

```
procedure squarify(list of real children, list of real row, real w)
begin
    real c = head(children);
    if worst(row, w) ≤ worst(row++[c], w) then
        squarify(tail(children), row++[c], w)
    else
        layoutrow(row);
        squarify(children, [], width());
    fi
end
```

During the construction of the data structure tree, a value is assigned to each node, which represents its weight compared to the other nodes. The weight of each node is the sum of the weights of its children. It is calculated from the bottom nodes to the root of the tree. In this way, the root node obtains the height weight, which represents the area of the initial rectangle.

Figure 6.7 shows the process of drawing a treemap from an example tree data structure according to the squarification algorithm. The first drawing step uses the root of the tree to draw the initial rectangle. Then, the algorithm draws the children sequentially until all the nodes are reached. At each drawing level, the algorithm is applied on the sibling nodes.

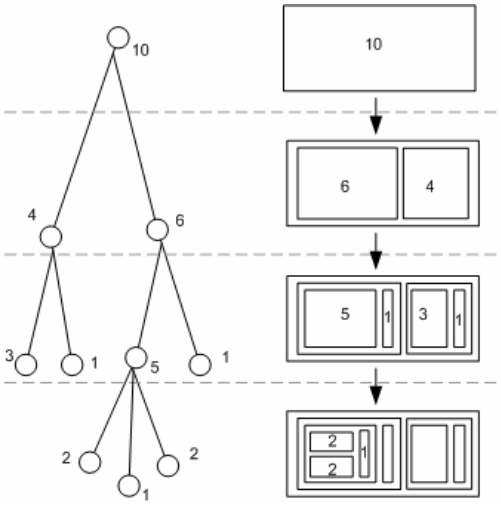


Figure 6.7: drawing the squarified Treemap from the tree data structure

6.3 Implementation

The class diagram of the new visualization is depicted in Figure 6.8, wherein the main classes and their interactions are shown. The class diagram is the same for both visualization techniques, because those techniques are only different in their graphical views (see Figure 6.3) and this difference can be achieved by adjusting the content of the xLayout class and parts of other classes. The letter 'x' in the class name refers to Tree or Treemp.

The Class xPlugin creates the graphical user interface components. It initiates the parser represented by class TreeBuilder that creates the data table and the tree data structure from the Rstore dataset. Class xContoller checks whether the input fits the visualization and creates a new windows using class xWindows.

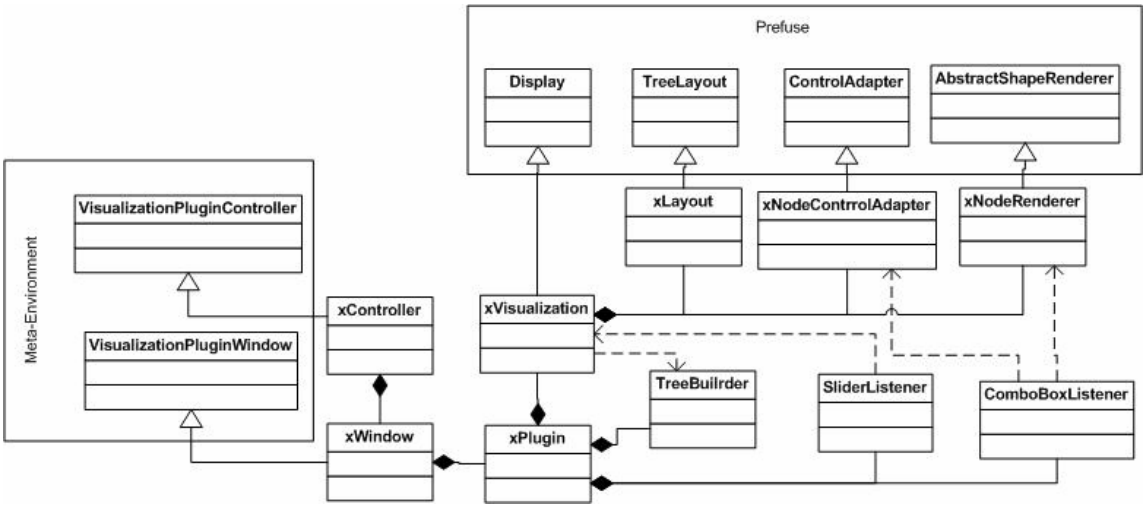


Figure 6.8: class diagram of the two new visualization techniques, x = Tree or Treemap

The created tree data structure is used in the class xVisualization to build the display using the helper classes. Class xVisualization is an extension of Prefuse class Display. This class is responsible for initiating the display and assigning controllers. Another important task of this class is updating the treemap cell padding and repainting the display.

```
public void updatePadding(int padding){
   m_padding = padding;
   treeMapLayout.setFrameWidth(m_padding);
   this.update();
}
public void update(){
   this.repaint();
}
```

Class TreeBuilder creates the tree data structure from a set of file paths defined in the input Rstore. It initiates the root, creates a set of nodes from each file path and adds them to the tree. Then, it removes the first nodes that have no more than one child. Finally, it sets the attributes of each node.

```
public Tree buildTreeFromRStore(RTuple fact){
    try {
        RElem set = fact.getValue();
        RElemElements elements = set.getElements();
        makeNewTree();
        setTreeNodes(elements);
        removeNodesUpToSystem();
         setOptionalNodeAttributes(elements);
     } catch (Exception e){
          e.printStackTrace();
     }
  return m_dataTree;
}
```

Class xNodeRenderer creates the graphical view according to the specification described in xLayout. The class xLayout arranges the positions of the visualization elements; therefore, it defines the layout type of the visualization. In case of the Treemap, it determines the size of the nodes and places them inside each other recursively, based on the hierarchical position defined in the data structure tree. The function *worst()* is implemented in the class xLayout as follow:

```
private double worst(List rlist, double w) {
   double rmax = Double.MIN_VALUE, rmin = Double.MAX_VALUE, s = 0.0;
   Iterator iter = rlist.iterator();
   while ( iter.hasNext() ) {
      double r = ((VisualItem)iter.next()).getDouble(AREA);
      rmin = Math.min(rmin, r);
      rmax = Math.max(rmax, r);
      s += r;
   }
   s = s*s; w = w*w;
   return Math.max(w*rmax/s, s/(w*rmin));
}
```

In order to be able to represent each software metric with the node size, the selected metric value of the node must be obtained from the data structure. The size of each parent is recursively determined by summing the software metric values of its children. To avoid hidden nodes, the node size less or equal than 0 will be set to 0.1.

```
iter = new TreeNodeIterator(root, false);
while ( iter.hasNext() ) {
   NodeItem n = (NodeItem)iter.next();
   double area = 0;
   if ( n.getChildCount() == 0 ) {
       area = n.getDouble(m_selectedMetric);
       if(area <= 0 ){ area = 0.1;}
   } else if ( n.isExpanded() ) {
       NodeItem c = (NodeItem)n.getFirstChild();
       for (; c!=null; c = (NodeItem)c.getNextSibling()) {
                   area += c.getDouble(AREA);
       }
   }
   n.setDouble(AREA, area);
}
```

The xNodeRenderer class is responsible for drawing the shape of the nodes according to the node type, which are file or directory, that is defined in the tree data structure. In case of the navigation tree, directories are drawn as folder icons and the files have rectangular shapes with different heights depending on their sizes. The class xNodeRenderer is also responsible for the color of the node as well as the node texture, text, and its intensity. Hence, it contains the nest functios:

```
private void fillNodes(Graphics2D g, VisualItem item, Shape shape)
private void fillLinks(Graphics2D g, VisualItem item, Shape shape)
private void fillTexture(Graphics2D g, VisualItem item, Shape shape)
private void drawText(Graphics2D g, VisualItem item, Shape shape)
private void animateBorder(Graphics2D g, VisualItem item, Shape shape)
protected Shape getRawShape(VisualItem item)
```

Class xNodeControlAdapter handles the interaction by reacting to the events produced by focusing or clicking the nodes. Each node has its renderer, by which it is possible to repaint one node individually and leave the rest unchanged. The whole display can also be repainted. The hierarchical space can also be increased or decreased with a slider. The events that have been introduced by adjusting the slider are handled by class SilderListener, which result in repainting all of the display. The node color, texture, and labels can be adjusted with a number of combo boxes. Class xNodeControlAdapter contains:

```
public void itemClicked(VisualItem item, MouseEvent e)
public void itemEntered(VisualItem item, MouseEvent e)
public void itemExited(VisualItem item, MouseEvent e)
private void removeAllLinks(Node nextNode)
private Node getLastPathNode(Tree tree, String filePath)
private Color getComplexityColor(String[] methods, int lineNumber, int i)
private Style getOverviewStyle(StyledDocument doc, Color color)
private int getColorIndex(String[] methods, int lineNumber)
```

Distinguishing the nodeRenderer from the ControlAdapter leads to complications in updating the nodes. The events produced by the combo boxes have to be sent to both classes xNodeRenderer and xNodeControlAdapter, otherwise repainting the nodes with different colors does not work, a Prefuse drawback.

6.4 TreemapVis

The derived visualization technique is shown in Figure 6.9, which consists of two parts, a squarified treemap and code overview viewer, in addition to a setting panel. The squarified Treemap represents the file system hierarchy of the program. Its leaf rectangles represent the files, which are drawn inside larger rectangles that represent the directories. The biggest rectangle that contains all other rectangles is the main directory of the program under investigation.

There are four fundamental visualization parameters associated with each treemap rectangle that represents a program file: size, color, textual value and texture. Each of these parameters can represent a specific software metric selected by the user through the selection boxes. For example, the size can represent the line of code (LOC); the greater the area is, the larger the LOC will be. As for the color parameter, the darker the color, the higher the metric value and similarly the darker the texture, the higher the metric value. Specific software metrics can also be displayed with extra parameters. For example, LOC can be shown with all parameters at the same time.

According to a number of papers, Treemap is not optimal for displaying hierarchy and groups. Therefore, this new visualization provides the option to show hierarchical paths. The hierarchy path can also be hidden to save display space and use it for the leaf rectangle. The directories in the software system are drawn in green, where the color represents the hierarchy depth level. The lighter the green is, the deeper the hierarchy will be.

Each treemap rectangle that represents a file is linked with its associated source code. An overview of the source code is viewed on the right panel and will be activated by focusing on a file node. The source code overview is an abstraction of the code used to view the metric values of parts of the source code such as the metrics function. By moving the mouse over the rectangles, the code belonging to the rectangle will be shown on the viewer. Focusing on a file node is indicated with highlighting of the rectangle and viewing a tooltip with the rectangle name. The overview viewer can be frozen by clicking on a rectangle by which the code belonging to that rectangle is shown. In addition, the detailed source code can be displayed by double clicking on the code overview.

Each line of the source code is represented as a single colored line on the right panel. Line length indicates the length of the line and indentation in the source code. Thus, it enables a rough control structure to be viewed on the screen. Furthermore, line color can be used to represent a variety of aspects represented by metrics software.

In this technique, the type of the metrics belonging to the files is open and the metrics set is scalable. Representation of software metrics with the visualizable parameters is optional. In addition, viewing the code overview and software metrics is also optional since an Rstore with a set of system files are enough to draw a basic Treemap.

A display of all connected file(s) with any specific file can be reached by right clicking on that specific file. This action will paint that specific file rectangul in black and the related file rectangles in dark grey. A second right click anywhere in the display view will cancel this action. To speed up the search activity, there is also a search field available, which makes searching for a specific file or directory possible. The files or directories found will be painted in blue. Beside the representation of the information with visualization parameters, the information is also shown textually on the right panel.

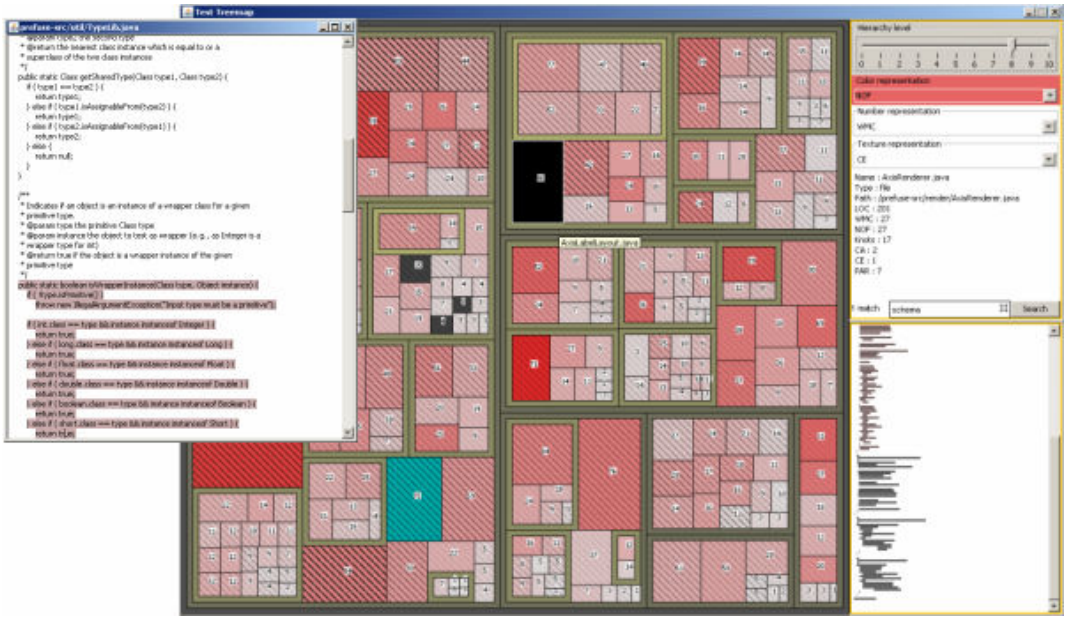


Figure 6.9: a screenshot of TreemapVis

6.5 TreeVis

TreeVis, the second visualization shown in figure 6.10, combines the Navigation tree with the overview viewer. The Navigation tree is not a compact visualization as is Treemap, but it provides other features that the Treemap does not have. Since the nodes of a navigation tree are drawn separated from each other, it is possible to place another layer to draw the links between the files without making the visualization complex. With the Navigation tree it is also possible to use more color parameters to represent more software metrics while multiplying the colors in Treemap rectangles makes the rectangles less recognizable. Another advantage of the Navigation tree is its support for filtration by which subdirectories can be hidden. Filtration of the non-relevant parts of the file system would help the user to focus on the important part. In addition, it supports moving and zooming the entire tree.

TreeVis uses the same Rstore data format as the TreemapVis, therefore the same software system can be visualized with TreeVis as well as with TreemapVis, and both visualizations can be generated and opened beside each other. It can be used in the same way as TreemapVis and it has exactly the same functionalities. The code overview viewer displays an overview of the code associated with the file focused under the mouse. A sub tree can be hidden or shown by clicking on the parent directory and the whole tree can be expanded using the slider on the setting panel.

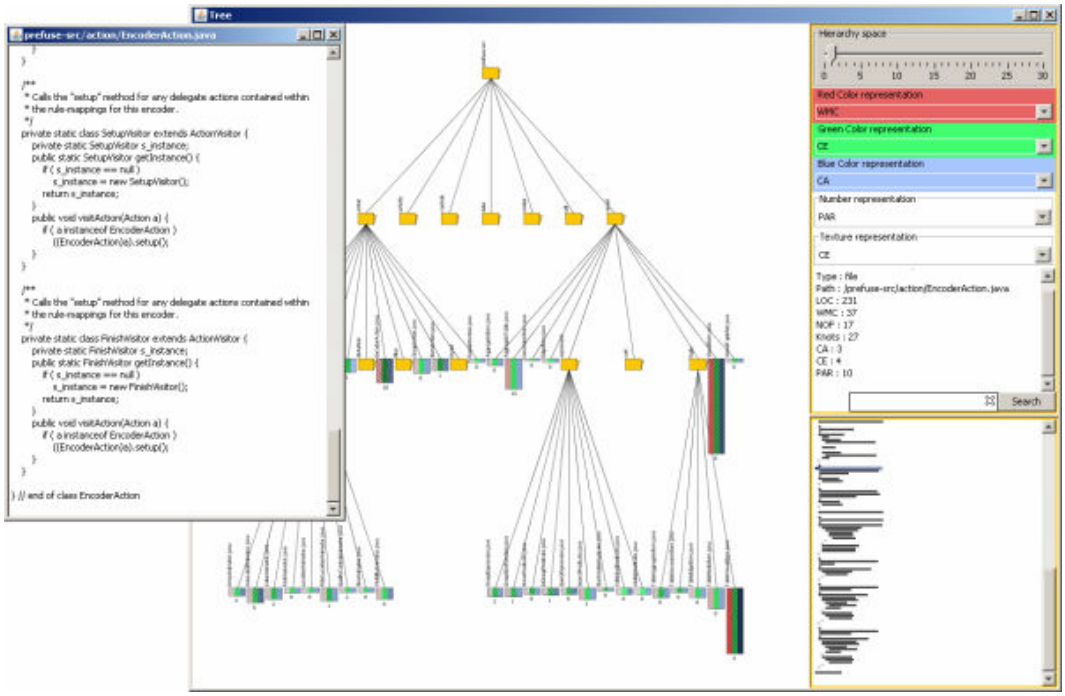


Figure 6.10: a screenshot of TreeVis

7 Experiments and discussion

In order to examine the improvement of software comprehension through using the two new visualization techniques, the new techniques have been compared against a table-view that includes search and sort functionality. Microsoft Excel or Open Office have been chosen as alternatives for the table-view, since the Meta-Environment provides a functionality to export the table with results of software analysis as a CSV file, which can be viewed using these tools. The comparison is carried out by answering a set of questions devised according to Figure 2.1, which explains how visualization supports the cognitive process.

The aim of the experiment is to determine whether solving tasks in a given system differs with respect to task completion times by using the visualization versus the table-view. Therefore, the null hypothesis is supposed that there are no such differences. The purpose is not to compare all the characteristics of the techniques against each other, but rather to use observed quantitative differences as an indication for interest in a subsequent qualitative analysis.

In the next section, a set of questions is given that satisfy the methods described in Chapter 2. To avoid a long list of questions, a number of questions are grouped to satisfy multiple methods. Each question should be answered twice: with one of the new visualization techniques and with the table view using Open Office or Excel. In order to make the test-results comparable, the same dataset has been used for both of the visualization techniques and the table view. However, the values of the dataset have been adjusted to avoid using results found with one technique in the other technique.

Increased resources: Parallel processing

1. Find the number of files that are linked with the largest file (having the highest LOC)?

Offload work from cognitive to perceptual system & External memory

2. Find the largest file that is linked with the file having the highest WMC?

Increase storage and accessibility

3. How many files are in the directory "/prefuse-src/action/layout/"?

Reduced search: Grouping & Enhanced recognition of patterns & High data density

4. How many files have MWC higher than 70 and LOC smaller than 200?

Structure (Spatially indexed addressing)

5. How many files in the directory "/prefuse-src/util/force" are linked with files outside the directory?

Abstraction and aggregation Visual schema for organization

6. Do all directories include complex files (with WMC higher than 50)?

Perceptual interface (Visual representations make some problems obvious)

7. Is coupling among the different directories high or low? (combination of CA and CE indicates directory coupling, CA is the number of depended packages, CE is the number of packages that the file depends on)

Perceptual monitoring & Manipulating medium

8. Find begin and end line number of the most complex method in the file with highest "Knot" value? (the table view does not provide the possibility to display the file content, therefore the files must be explored manually)

In this experiment, 11 computer science or software engineering students participated. The time measurement was controlled by a second person. The time for reading and understanding the questions was excluded. Furthermore, all of the participants were asked to practice all of the functionalities of the visualization techniques as well as the search and sort functionalities in MS Excel or OpenOffice in advance. It is assumed that the experience needed to work with MS Excel or OpenOffice is nearly the same as that needed to work with the new visualization techniques.

Due to the relatively small number of participants, hard conclusions cannot be drawn. However, the results provide a global impression of the improvement rate. From the results of the experiments (see appendix C), three histograms have been created. The first two diagrams, 7.1 and 7.2, represent the total answering time per participant using the two visualizations versus table-

view.

From the diagrams, it can be observed that the total time needed for answering all questions has been reduced compared to the table-view. It is clear from Figure 7.1 that TreemapVis reduces the required time by about 50% (from 941.6 seconds to 432.50 seconds). Due to the difference in experience using visualization, sort and search functions and the reaction time of the participants, the reaction time of each participant was different. Nevertheless, it is clear from the results that using TreemapVis has improved the participant answering speed significantly.

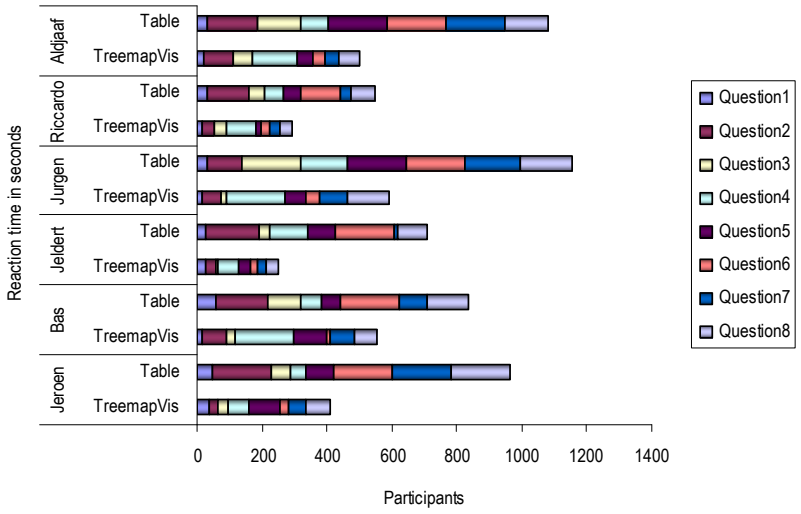


Figure 7.1: Answering speed per user using TreemapVis

In Figure 7.2, Treevis reduction of the total time needed for answering the questions was almost negligible. Undoubtedly, the reason is the incompactness of the Treevis technique, which causes the users to move the tree, search for the nodes, and to zoom in to navigate for details. See 7.6 for more details.

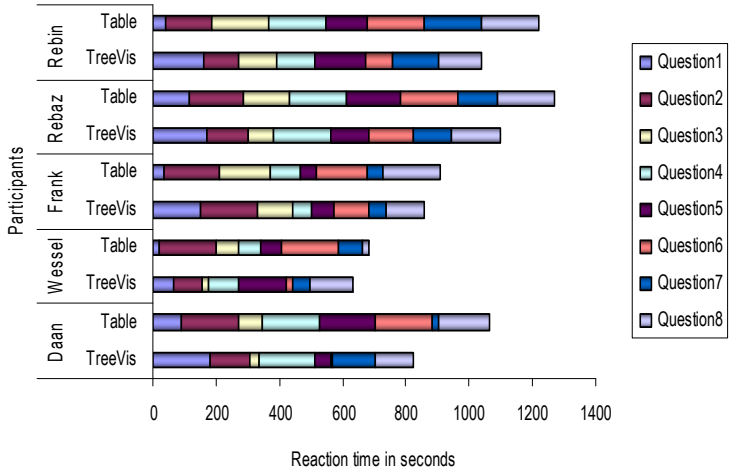


Figure 7.2: Answering speed per user using Treevis

Figure 7.3 illustrates the reaction time per question using the three techniques. It is clear that TreemapVis delivers a significant improvement for all questions except question number 4, while TreeVis deteriorates the results of table view in most of the questions. Considering the difference in results for the first question, compactness of the visualization seems to be helping the user in parallel processing. Because the user does not need to move and zoom the display to find needed information, answering the question with TreemapVis requires less time than with the table view, while with TreeVis it requires more time.

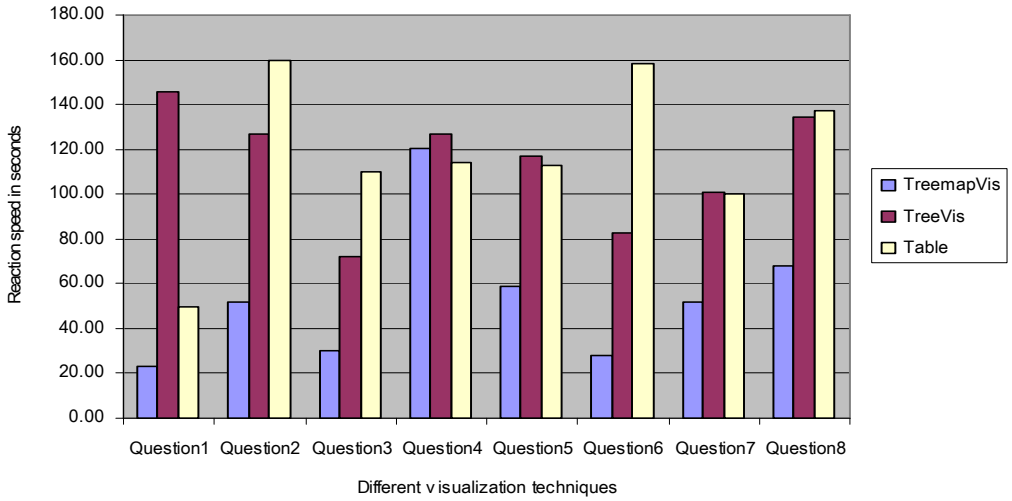


Figure 7.3: the mean answering time per question with the three techniques

The time required for answering questions 2 and 3 increased by using both visualizations. Therefore, it can be said that the two techniques, especially TreemapVis, can aid as an external memory that simplifies the accessibility. Surprisingly, none of the two visualizations help in accelerating the answer to question 4. Combining a number of node attributes is required to answer the question, where one of the attributes must be used to determine a boundary, and the second attribute must be searched. It was not easy for the users to visually distinguish nodes within this boundary. The reason is a mistake in the detail design and can be solved by adding the functionality to filter the nodes above or below a specific range. The filtration can be done by coloring the uninteresting node within a range that the user can set. This will reduce the number of nodes and makes it easier to determine nodes inside a specific boundary.

Furthermore, as it seems from question 6, TreemapVis provides a clear overview of the system, which enhances abstraction and aggregation significantly. Answering the question requires looking at all of the nodes and comparing their directories. From the last two questions, it appears that TreeVis provides no improvement while TreemapVis improves the results significantly.

From the observation of test execution and the feedback of the participants, the following points can be derived:

- Filtration functionality is required, by which the user can hide files and directories having smaller attribute values than a specific limit.
- It is easier to recognize and distinguish the node attribute values if the user can adjust the minimum and maximum values for the color hue of the nodes.
- Using the right-click to view and hide the links between the files is confusing for some users.
- Almost no users use the texture element. The possible reason is the texture's unclarity and uncertainty.
- Apparently each user develops his own strategy for answering the questions. Therefore, with more experience, the questions could be answered faster. This is the same for MS Excel or OpenOffice.

Unfortunately, there was not enough time to add the two functionalities described in the first two points to the implemented technique, and therefore it is considered as a further task.

8 Conclusion and further work

In this research, an investigation has been carried out to conclude a number of suitable visualization techniques for the Meta-Environment with the aim to improve the impact of the current table-view on software understanding.

This research was accomplished in a number of steps. Initially, the limitation of Rscript was studied, from which it appeared that almost all static visualization techniques are modelable using the Rscript language, while visualizations that require real values as basic design elements were harder to model due to missing support for numeric real type.

Furthermore, based on an introduced model for software visualization fields, static program visualization has been selected to be the main field of this research, since maintenance is the main application field of the Meta-Environment and static visualization techniques are suitable to use in this field and for this research.

From the requirements and usage of the Meta-Environment, it was apparent that the new visualization technique must be able to visualize the common features of large software systems, especially legacy systems. For such a generic visualization technique, the optimal properties have been denoted such as graphically visualizing the source-code file hierarchy, the relation between the files and the properties of each file, in addition to linking the visualization elements to the source code. Based on resources, selection criteria have been suggested for such a 'software exploration technique', as it is called.

In addition, an overview of 2-dimensional static program visualization techniques has been given. As a result of the evaluation of appropriate techniques in the overview, the squarified treemap has been selected to be the main part of the exploration tool. For comparison purposes, a navigation tree has been selected as the main component of an additional visualization technique.

By using the Prefuse reference model, the new techniques have been designed and implemented resulting in two new visualization techniques, TreemapVis and TreeVis. During the design phase, attention has been paid to the common visualization problems as well as the specific treemap problems. Consequently, viewing of the hierarchical treemap has been made flexible and optional.

In addition to the main objective of this research, the software visualization terms and definitions, such as software visualization, program visualization, visualization type, and visualization technique, have been clarified. The relationship between program comprehension and program visualization has been discussed and a list of potential factors for software comprehension difficulties has been proposed. Furthermore, a classification of static program visualization has been introduced based on display mode.

Results of experimentation of the two new visualizations reveal that TreemapVis is significantly better than the table-view to use for understanding results of software analysis in its application domain, while the improvement of TreeVis is negligible. TreemapVis reduces the time and effort required for understanding software analysis results by more than 50%, which makes it a good solution of the research problem. Adding filtering functionality of the nodes and the values of their attributes within a specific limit can increase the improvement even more.

It can be concluded that the compact representation of visualization elements is crucial for the visualization technique especially in the case of visualizing large and legacy systems. Furthermore, the right choice of color range, visualization elements and the number of colors plays an important role. In addition, it appears that texture is less effective than text, size and color as a basic visualization element.

Despite the fact that no software visualization technique exists as a generic alternative for tables, the new visualization techniques have been made as generic as possible, to be used for all software systems with different sizes and styles and act as a good alternative for the table view in its domain application.

In further work, a research should be carried out to develop a tool within the Meta-Environment that can extract the required data from the investigated software for the new exploration tool.

Bibliography

[1] Corbi, T.A., Program Understanding: Challenge for the 1990s, IBM Systems, Journal, Vol. 28, pages 294-306, 1989.

[2] Padula, A., Los Alamitos, Use of a Program Understanding Taxonomy at Hewlett-Packard, in Proceedings 2nd Workshop on Program Comprehension, pages 67-70, IEEE Computer Society, 1993.

[3] Von Mayrhauser, A. Vaus, A.M. On the role of program understanding in re-engineering tasks, Aerospace Applications Conference, 1996. Proceedings.,1996 IEEE, vol.2, pages 253-267, 1996

[5] Anneliese von Mayrhauser**,** A. Marie Vans, Program Comprehension During Software Maintenance and Evolution". IEEE Computer, pages 44-55**,** 1995

[6] E. B. Swanson B. P. Lientz and G. E. Tompkins, Characteristics of application software maintenance, Communications of the ACM, vol. 21, pages 466-471,1978

[7] K.K Holgeid, A study of Deevelopment and Maintenance in Norway: Assessing the Efficiency of Infomation Systems Support Using Functional maintenance, Information and software technology, Vol. 42, N. 10 , pages 687-700, 2000

[8] Tom Erkkinen, Maurice F. Snyder, Reducing Software Maintenance Costs By Designing in Reliability, Applied Dynamics International, Inc

[9] Wen Jun Meng, Jürgen Rilling, Yonggang Zhang, René Witte, Philippe Charland, An Ontological Software Comprehension Process Model, Proceedings of the ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML 2006), Springer, LNCS, 2006.

[10] K. Kowalczykiewicz and D. Weiss. Tracebility: Taming uncontrolled change in software development. Proceedings of IV National Software Engineering Conference, Tarnowo Podgorne, Poland, 10 pages, 2002

[11] Michael P. O'Brien, Software Comprehension – A Review & Research Direction, Technical Report UL-CSIS-03-3, University of Limerick, November 2003.

[12] Russell Wood, Assisted Software Comprehension, Final report, June 2003

[13] Diehl. Stephan, Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software, ISBN 3540465049, XII, Springer, 2007

[14]Deimel, L., Naveda, J., Reading Computer Programs: Instructor's Guide and Exercises, Technical Report CMU/SEI-90-EM-3, Software Engineering Institute, Carnegie Mellon University, 1990

[15] Steven P. Reiss, Bee/Hive: A Software Visualization Back End, Proceedings of ICSE 2001 Workshop on Software Visualization, Toronto, Ontario, Canada, pages 44-48, 2001

[16] Johnson, W.L.; Erdem, A, Interactive explanation of software systems, Knowledge-Based Software Engineering Conference, Issue , 12-15 Nov 1995, pages 155 – 164, 1995

[17] Storey, M.-A. , Theories, Methods and Tools in Program Comprehension, Program Comprehension, 2005. IWPC 2005. Proceedings 13th International Workshop on volume , Issue , 15-16 May 2005, pages 181 – 191, 2005

[18] James Westland Cain Rachel Jane McCrindle, Software Visualisation using C++ Lenses, Program Comprehension,Proceedings 1999 IEEE, Seventh International Workshop,pages 20-26, 1999

[19] Gabriel, R.P., Patterns of Software, Tales from the Software Community, ISBN : 0195121236, Oxford University Press, 1996.

[20] Soukup, J., Taming C++: Pattern Classes and Persistence for Large Projects*,* ISBN 0201528266*,* Addison Wesley, 1994

[21] PETRE, M; QUINCEY, E. A gentle overview of software visualization. PPIG Newsletter – September 2006.

[22] Kirsten R. Butcher, Cognitive Processes and Visualization, Teaching Geoscience with Visualizations: Using Images, Animations, and Models Effectively. Carleton College , Northfield , MN . 2004

[23] Stuart K. Card, Reading in information visualization: Using Vision to Think, Morgan Kaufmann Publishers, San Francisco, ISBN 1558605339, 1999, 686 pages

[24] J. Stasko, J. Domingue, M. Brown, B. Price, Eds, Software Visualization: programming as multimedia experience, MIT Press,

1998.
[26] Melanie Tory, Torsten Möller, Human Factors in Visualization Research, IEEE Transactions on Visualization and Computer Graphics, Volume 10 , Issue 1 (January 2004), Pages 72-84, 2004
[27] C. Knight, Visualisation for Program Comprehension: Information and Issues, Department of Computer Science, University of Durham, Technical Report 12, 1998
[28] Price, Baecker and Small, A Taxonomy of Software Visualization, Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences, vol 2, pages 597-606, 1992
[29] Stefan-Lucian Voinea, Software evolution visualization, proefschrift, PhD Thesis, Technische Universiteit Eindhoven, 2007
[30] Cleveland. W. S., and Mcgill, R. Graphical perception: Theory, experimentation and application to the development of graphical methods. J. Am. Stat. Assoc. 79, 387, 531-554, 1984
[31] Mackinlay, J., Automating the Design of Graphical Presentations of Relational Information, ACM Transactions on Graphics. 5(2, April), 110-141, 1986
[32] Adrian Ulges, Visualizing Software Evolution, University of Kaiserslautern, Seminar "Software Visualization", October 29, 2005
[33] Andrian Marcus, Louis Feng, Jonathan I. Maletic, 3D Representations for Software Visualization
[34] M. G. J. van den Brand, M. Bruntink, G. R. Economopoulos, H. A. de Jong, P. Klint, T. Kooiker, T. van der Storm, J. J. Vinju, Using The Meta-Environment for Maintenance and Renovation, Proceedings of the 11th European Conference on Software Maintenance and Reengineering, Pages 331-332,2007
[35] Paul Klint, Rscript --- a Relational Approach to Software Analysis, DRAFT: 2nd May 2005
[36] P. Klint, A Tutorial Introduction to RScript--a Relational Approach to Software Analysis, draft, pdf, 2005
[37] Prefuse user;s manual, http://prefuse.org/doc/manual/introduction/structure/
[38] Teemu Rajala, Mikko-Jussi Laakso, Effectiveness of Program Visualization: A Case Study with the ViLLE Tool, Journal of Information Technology Education, vol 7, 2008
[39] Storey, Margaret-Anne, A cognitive framework for describing and evaluating software exploration tools, PhD thesis, School of Computing Science, Simon Fraser University. 1998,
[40] Schnotz, W., Bockheler, J. & Grzondziel, H.. Individual and co-operative learning with interactive animated pictures. European Journal of Psychology of Education, 14 (2), 245-265, 1999
[41] Tversky, B.; Morrison, J.B.; and Betrancourt, M., .Animation: can it facilitate? International Journal of Human-Computer Studies, 57, 4, pages 247-262, 2002
[42] B.B. de Koning, Effects of cueing and interactivity in dynamic visualizations, project, 2006-2010
[43] Sandro Boccuzzo, Harald Gall, "CocoViz: Towards Cognitive Software Visualizations," vissoft, pp. 72-79, 2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis, 2007
[45] Michele Lanza, Stephane Ducasse, The Class Blueprint: Visually Supporting the Understanding of Classes, IEEE Transactions on Software Engineering, vol 31, issue 1, pages 75-90, 2005
[46] Margaret-Anne Storey, Casey Best, Jeff Michaud, SHriMP Views: An Interactive Environment for Information Visualization and Navigation, Conference on Human Factors in Computing Systems, CHI '02 extended abstracts on Human factors in computing systems, Pages: 520-521, 2002
[47] Jean-Marie Favre, Visualization of Component-based Software, Visualizing Software for Understanding and Analysis, 2002. Proceedings. First International Workshop, pages 51-60, 2002
[48] Alfred kobsa , User Experiments with Tree Visualization Systems, Information Visualization, 2004. INFOVIS 2004. IEEE Symposium, Pages 9-16, 2004
[49] John Lamping, Ramana Rao, and Peter Pirolli, A Focus+Context Technique Based on Hyperbolic Geometry for Visualizing Large Hierarchies. , Proceedings of the SIGCHI conference on Human factors in computing systems, pages 401-408, 1995
[50] J. Carriere and R. Kazman Research Report: Interacting with Huge Hierarchies: Beyond Cone Trees. In: Proceedings of IEEE Symposium on Information Visualization, Los Alamitos, pp. 74- 81, 1995.

[51] Luc Beaudoin, Marc-Antoine Parent, Louis C. Vroomen, Cheops: A Compact Explorer For Complex Hierarchies, IEEE Visualization, Proceedings of the 7th conference on Visualization '96, San Francisco, pages 87-ff, 1996
[52] Soon Tee Teoh and Kwan-Liu Ma , RINGS : A Technique for Visualizing LargeHierarchies, Lecture Notes In Computer Science; Vol. 2528, Pages 268-275, 2002
[53] Mark Bruls, Kees Huizing, and Jarke J. van Wijk , Squarified treemaps, Eindhoven University of Technology. Dept. of Mathematics and Computer Science
[54] Taowei David Wang1, Bijan Parsia, CropCircles: Topology Sensitive Visualization of OWL Class Hierarchies, 5th International Semantic Web Conference, Athens, GA, USA, November 5-9, 2006,
[55] Kai Wetzel pebbles, using Circular Treemaps to visualize disk usage, http://lip.sourceforge.net/ctreemap.html
[56] Michael Balzer, Oliver Deussen, Claus Lewerentz Voronoi Treemaps, Proceedings of the 2005 ACM symposium on Software visualization, Pages 165-172, 2005
[57] Todd Barlow Padraic Neville, A Comparison of 2-D Visualizations of Hierarchies, Information Visualization, INFOVIS 2001n IEEE Symposium, pages 131-138, 2001
[58] Yipeng Li and Boonth Nouanesengsy, Autumn quarter, 2004, Improved Navigation for TreeRing, Term-project proposal for CSE788 Information Visualization, 2004
[59] Eick, S.C.; Steffen, J.L.; Sumner, E.E., Jr, Seesoft-a tool for visualizing line oriented software statistics, Software Engineering, IEEE Transactions, vol 18, issue 11, 1992, pages 957-968, 1992
[60] Boonthanome Nouanesengsy, Yipeng Li , Hierarchy Visualizations, 2004
[61] Frank van Ham, Using Multilevel Call Matrices in Large Software Projects, Proceedings of the IEEE Symp. Information Visualization, IEEE Press: New York, 227-232, 2003
[62] Controlling architecture with structure101, White paper, www.headwaysoftware.com, july 2007
[63] Shengdong Zhao, Michael J. McGuffin, Mark H. Chignell, Elastic Hierarchies: Combining Treemaps and Node-Link Diagrams, Information Visualization, 2005. INFOVIS 2005. IEEE Symposium, pages 57-64, 2005
[64] Danny Holten , Hierarchical Edge Bundles:Visualization of Adjacency Relations in Hierarchical Data, IEEE Transactions on Visualization and Computer Graphics, vol 12, issue 5, pages 741-748, 2006
[65] Veit Botsch and Christoph Kunz, Visualization and navigation of networked information spaces: the Matrix Browser, Information Visualisation, 2002. Proceedings. Sixth International Conference, pages 361-366, 2002
[66] Maurice Termeer Christian F. J. Lange Alexandru Telea Michel R. V. Chaudron, Visual Exploration of Combined Architectural and Metric Information, Proceedings of the 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis, page 11, 2005
[67] J.-D. Fekete, D. Wang, N. Dang, A. Aris, and C. Plaisant, Overlaying Graph Links on Treemaps. In Proceedings of the 2003 IEEE Symposium on Information Visualization (InfoVis'03), Poster Compendium, pages 82–83, 2003.
[68] Danny Holten, Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data, IEEE Transactions on Visualization and Computer Graphics, vol 12, issue 5, pages 741-748, 2006
[69] Sinha, V. Karger, D. Miller, R., Relo: Helping Users Manage Context during Interactive Exploratory Visualization of Large Codebases, IEEE Symposium on Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006, pages 187-194, 2006
[70] Hao Ming C., Dayal Umeshwar, Keim Daniel, Schreck Tobias, A visual analysis of multi-attribute data using pixel matrix displays, Proceedings of the SPIE, Volume 6495, pages 6495-04, 2007
[71] Jean-Daniel Fekete, Niklas Elmqvist, Thanh-Nghi Do, Howard Goodell, Nathalie Henry, Navigating Wikipedia with the Zoomable Adjacency Matrix Explorer, Technical report, INRIA Research Report (Paris), no RR-6163, 2007
[72] Jarke J. van Wijk, Huub van de Wetering, Cushion Treemaps: Visualization of Hierarchical Information, Proceedings of the 1999 IEEE Symposium on Information Visualization, page 73, 1999
[73] Turo D., Johnson B., Improving the visualization of hierarchies with treemaps: design issues and experimentation, Proceedings of the 3rd conference on Visualization '92, pages 124-131, 1992

[74] Benjamin B. Bendersom and Ben Shneiderman, Ordered and Quantum Treemaps: Making Effective Use of 2D Space to Display Hierarchies, ACM Trans. Graph., Vol. 21, No. 4. (October 2002), pp. 833-854, 2002
[75] Martin Wattenberg, Ben Bederson, Dynamic treemap layout comparison, http://www.cs.umd.edu/hcil/treemap-history/java_algorithms/LayoutApplet.html
[76] Steven P. Reiss, The Paradox of Software Visualization, 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis. VISSOFT 2005, pages 1-5, 2005
[78] St. Louis, Missouri, Visualizing multiple evolution metrics,Proceedings of the 2005 ACM symposium on Software visualization, pages 67-75, 2005
[79] Abello, J. van Ham, F., Matrix Zoom: A Visual Interface to Semi-external Graphs, IEEE Symposium on Information Visualization, 2004. INFOVIS 2004, pages 183-190
[80] Maurice Termeer, MetricView Visualization of Software Metrics on UML Architectures, 16th January 2005, http://www.win.tue.nl/empanada/metricview/
[81] Jean-Daniel Fekete, "The InfoVis Toolkit," infovis, pp. 167-174, IEEE Symposium on Information Visualization (INFOVIS'04), 2004

Appendix A: Example list of visualization modeling using Rscript

Relation	Representation	Visualizations
rel[str, str]	rel[file, file] : file dependency	1. Directed graph, the arrows represent the dependency 2. links with two different colors for the ends of the link
	rel[file, class] : classes per file (e.g. in java)	1. Class rectangles inside file rectangles 2. Tree: root represents the file and the children the classes.
	rel[class, method] : methods per class	1. Methode namen in vierkanten die class representeren 2. Tree
	rel[class, method] : class methods call	Directed graph, rectangles are the classes, ovals are the methods
	rel[file, variable] : variables per file	1. Variables in file rectangles 2. Tree
	rel[class, variable] : variables per classes	1. Variable names in rectangles 2.Tree
	rel[class, class] : class dependency	Directed graph
	rel[method, method] : method method call	1. Directed graph 2. Two different colors links
	rel[method, variable] : variables inside methods	1. Variable names in rectangles 2.Tree
	rel[method, variable] : input parameters per method	1. Variable names in rectangles 2. Directed graph
	rel[class, interface] : implement relation	1. Directed graph
	rel[class, class] : extend relation	1. Directed graph 2. shape containment
rel[str, int]	rel[file, LOC] : files line of code	1. textual in rectangles 2. color in rectangles 3. size of the rectangles 4. texture in rectangles
	rel[class, LOC] : class line of code	==
	rel[class, line method count] : lines per method	==
	rel[class, call count from] : calls per class	== Or directed graph
	rel[class, call count to] : calls per class	==
	rel[class, number of vars] : vars per class	==
	rel[class, complexity] : class complexity	==
	rel[class, characters count] : characters count per class	==
	rel[dir, file count] : files count per dir	==
	rel[dir, class count] : class count per dir	==
	rel[class, links count] : class cohesion	==
rel[loc, loc]	loc can represent file, class, method, and variables, therefore all the previous possibilities are suitable	
rel[loc]	rel[file] : file hierarchy, file path, names, LOC,	1. File hierarchy tree with file properties such as

	char count loc contains path, object name, object LOC, object chars count. The hierarchy can be constructed from rel[loc]	LOC, name char count represented with size, text, color and texture attributes 2. treemap
	The same as above for package, classes, methods, directories	==
rel[loc, set[loc]]	rel[file] : file hierarchy , their path, names, LOC, char count in addition to set of related files	The same with the links among the files
	The same as above for package, classes, methods, directories	==
rel[loc, rel[str,int]]	rel[file] : file hierarchy, file path, names, LOC, char count in addition to set of properties with their values	1. File hierarchy tree of file hierarchy with file properties such as LOC, name char count represented with size, text, color and texture attributes. Each file has a set of properties with their value 2. treemap
rel[loc, set[loc], rel[str, int]]	rel[file] : files, their path, names, LOC, char count set of related files per file, set of properties per file with their values	1. File hierarchy tree with file properties such as LOC, name char count represented with size, text, color and texture attributes with the links among the files, each file has a set of properties with their value 2. treemap
	The same for packages, class, methods,	==
rel[str, str, int]	rel[file, file, links count] : link count among classes	file graph with multiple links The link count can be represented with width, color and text
	The same for package, classes, methods and dir	==
	rel[file, class, LOC] : line count per methods per classes	Tree having different node type for files and classes, LOC of the classes can be represented with size, color, text or texture.
	The same for package-class, class-method, directory-file and directory-class	==
rel[str, str, str]	rel[class, class, method] : structure of classes and their methods	Graph where the leaves represent the methods
	The same for dir-dir-class, dir-class-methods, dir-dir-files, file-file-method, class-class-method, class-class-method, etc.	==
rel[str, str, str, int]	rel[class, class, method, LOC] : the same plus line count per method	Graph, leaves represent the methods, size, color, texture of the leaves represents its LOC
	rel[dir, class, method, complexity] : file structure dir, classes and their methods plus the complexity per method	==
rel[str, str, rel[str], int, int]	rel[package, package set[class], size, complexiteit] package structure with relations and their classes plus size en complexity	Graph, nodes are the packages which contain the inteir classes
	The same for file-file-set[method], size, complexity	==
rel[str, set[str]]	rel[file, set[file]] : link structure among system files	Graph
	The same for classes, packages, methods	

rel[str, str, set[str], int]	rel[dir, parent dir set[files], LOC] : system file hierarchy with dir and files specified plus the size of the files	1. Graph 2. Treemap
rel[str, str, set[str], int, int]	rel[dir, parent dir, set[files],size, complexity] : the same as above plus size and complexity of the leaf nodes (files)	1. Graph 2. Treemap Color and size of the leaves represent the LOC and complexity
rel[str, set[str], set[str]]	rel[dir, set[file], set[dir]] : dir-file hierarchy	Graph
	rel[dir, set[class],set[dir]] : dir-class hierarchy	Graph
rel[str, str, set[str], set[str]]	rel[class, parent class, set[implement class], set[contain class]] : class hierarchy	Class diagram
rel[str,str,set[str],set[str],int]	rel[class, extends class, set[implment class], set[contain class], int] : class hierarchy with complexity	Class diagram with color, size, or texture attribute for classes
rel[str, set[loc]]	rel[class, set[loc]] : code line viewer such as sesoft if set[loc] contains an ordered set of code line chars	Such as Seesoft, in addition to packages which can be derived from loc
rel[str, str, rel[int, int]]	rel[class, package, rel[lines chars, color]] : rel[lines chars, color] is an ordered set according to the first element.	==
rel[str, set[loc], int]	rel[class, set[loc]] : code line viewer such as sesoft if set[loc] contains an ordered set of code line chars, in addition to an attribute per line e.g. color of the line	= =

Appendix B: An introduced classification of program visualization types

Program visualization is a young research area; its terms, methods and classifications are not standardized yet. Without such classification, it is difficult to gain an overview of the visualization types and techniques, which helps looking for a suitable visualization. A classification helps also in discovering new visualization possibilities that are not discovered yet. Therefore, a new classification of program visualization has been proposed in this research (see figure B1). This new model classifies the visualization types defined in chapter 2 according to their common display mode features, and therefore it is based on the display mode.

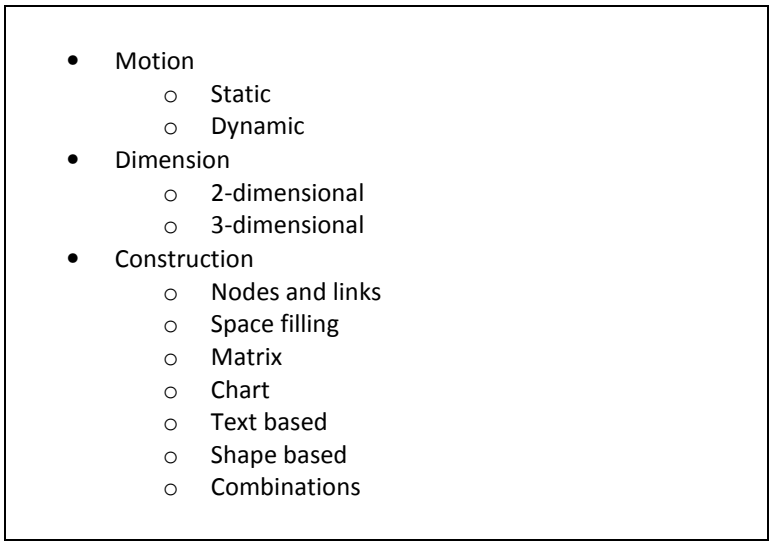


Figure B1: an introduced classification of program visualization techniques based on display mode

Every program visualization technique is visually static (motionless) or dynamic (be in motion). Static visualization techniques, such as architecture diagram, display the static aspects of software, while the dynamic visualizations such as algorithm animations visualize the dynamic behaviors of software. Static visualization aspects involve the program code, data structures, relation graphs, and organization of the program into modules. Program code is usually visualized as textual mode while the rest are visualized as graph. Examining the existing papers shows that most program visualization techniques used in the maintenance process are static while the most dynamic visualization techniques are used for debugging or understanding algorithms.

The existing program visualizations are normally 2 or 3-dimensional. Program visualizations having more than three dimensions do not exist, and single dimensional visualizations are meaningless. Most of the 3-dimensional program visualizations are the optimizations techniques of 2-dimensional version of the same visualization technique. An example is the botanical tree visualization, which is the 3-dimensional version of hierarchical tree.

The third aspect of the classification is related to the manner, by which the visualization is constructed from the visualization elements. Looking at the existing program visualization techniques, it can be realized that groups of them are sharing a number of common visual features. Based on these common features, a number of fundamental manners can be defined such as Nodes and links, Matrix and Charts. Other complex types can be formed by combining two or more of these fundamental types.

According to this classification, each program visualization technique has just one type of each motion, dimension and construction. For example, the Code-city visualization technique can be classified as 'static 3-dimensional space-filling' type and the class diagram as '2-dimensional static nodes-and-links' type.

The first construction type is Nodes-and-links, which refers to visualizing the information with interconnected nodes, which can have different shapes such as circles, rectangles and images. 2-dimensional node-and-link type involves a large number of

visualizations that can be organized in three different groups namely network, directed graph and tree.

The second construction type is space-filling, which aims to convey as much information as possible with as few pixels as possible [12]. Treemap is a typical example of space-filling type. Other fundamental type is Matrix, which uses a matrix to show the information such as dependency or evolution matrix. Charts within program visualization are mostly used for viewing software metrics or software evolution process, because it views relations or progresses among numeric values. Text-based visualizations mostly refer to highlighting the textual view of the code. Viewing textual information about the software aspects is also considered text-based visualization in this classification. Shape-based visualization in this classification refers to visualization that can not be covered by the previous types. An example is representing classes in object-oriented software with tables, which the length of legs of the table are representing specific software metrics [43].

Combination types are composed from combining two or more fundamental visualization types. The following is a list of possible combinations and their examples. For the types that do not exist in practice, possible examples are mentioned:

- Nodes and links - Space filling: call graph in treemap in which the treemap rectangles are graphs node [68].
- Nodes and links - Matrix: two hierarchy trees on top and left on evolution matrix [79].
- Nodes and links - Charts: Kiviat graph [78] or bar charts in nodes of class diagram [80]
- Nodes and links - Text based: TreeVis, the Meta-Environment's new visualization technique (see next chapter)
- Nodes and links - Shape based: perhaps does not exist, a possible example will be a call graph, which its nodes are house shapes and attributes of these houses represent software metrics.
- Space filling - Matrix: icicle tree combined with evolution matrix [81]
- Space filling - Chart: pixel matrix [70]
- Space filling - Text based: TreemapVis, the Meta-Environment's new visualization technique
- Space filling - Shape based: a possible example is an icicle tree with shapes for each node
- Matrix - Chart: dependency structure matrix
- Matrix - Text based: a possible example is dependency call matrix linked to the source code
- Matrix - Shape based: evolution matrix [13]
- Chart - Text based: a possible example is source code linked to changes per time as line chart
- Chart - Shape based: a possible example is code evolution line chart with shapes on the line
- Text based - Shape based: a possible example is code city overview linked to the source code
- Multi combinations: Argus viewer in which process explorer, dynamic sequence diagram and source code highlighting are associated in one display.

Appendix D: Results of experiments

Reaction speed in seconds of TreemapVis versus Table

	Jeroen		Bas		Jeldert		Jurgen		Riccardo		Aldjaaf	
	TreemapVis	Table	TreemapVis	Table	TreemapVis	Table	TreemapVis	Table	TreemapVis	Table	TreemapVis	Table
Question1	38	49	16	56	29	29	14	30	18	31	23	34
Question2	27	180	73	160	27	160	63	110	35	128	88	153
Question3	29	59	30	103	8	33	16	180	39	48	58	132
Question4	66	49	180	65	66	117	180	145	91	61	140	86
Question5	93	86	101	58	35	86	63	180	13	49	48	180
Question6	28	180	11	180	21	180	42	180	29	126	38	180
Question7	54	180	73	87	26	13	87	168	31	31	42	180
Question8	74	180	67	129	39	92	125	161	36	76	65	138

The max value 180 means more than 3 minutes. The orange value: impossible is assumed as more than 180 second

Reaction speed in seconds of TreeVis versus Table

	Daan		Wessel		Frank		Rebaz		Rebin	
	TreeVis	Table	TreeVis	Table	TreeVis	Table	TreeVis	Table	TreeVis	Table
Question1	180	90	64	19	150	33	173	115	160	40
Question2	127	180	91	180	180	180	126	170	110	145
Question3	28	75	19	71	113	160	82	148	120	180
Question4	177	180	98	70	58	93	180	180	120	180
Question5	50	180	150	65	70	50	120	169	160	132
Question6	5	180	22	180	110	163	142	180	89	180
Question7	133	20	53	75	55	50	120	127	145	180
Question8	122	160	136	23	120	180	158	180	135	180

Means

	TreemapVis	TreeVis	Table
Question1	23.00	145.40	49.82
Question2	52.17	126.80	159.75
Question3	30.00	72.40	109.65
Question4	120.50	126.60	113.88
Question5	58.83	116.67	112.85
Question6	28.17	82.33	158.56
Question7	52.17	101.20	100.12
Question8	67.67	134.20	136.97
Sum	432.50	905.60	941.6